Database System Development for Silicon Photonics

Tapio Koukkari

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 15.11.2017

Thesis supervisor:

Prof. Zhipei Sun

Thesis advisor:

D.Sc. (Tech.) Timo Aalto

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Tapio Koukkari

Title: Database System Development for Silicon Photonics

Date: 15.11.2017 Language: English Number of pages: 6+47

Department of Micro- and Nanosciences

Professorship: Photonics

Supervisor: Prof. Zhipei Sun

Advisor: D.Sc. (Tech.) Timo Aalto

Silicon photonics is a rapidly developing field of research. It is used in many applications, such as database interconnections. Currently the Integrated Photonics research group at VTT produces a larger amount of research data due to surging interest in silicon photonics. To better handle growing information resources, a database system was designed and developed in this thesis. The database used in the thesis project was MongoDB, a free and open-source cross-platform document-oriented database program. MongoDB was chosen mostly due to its flexibility which was deemed useful in a swiftly developing research area. An application layer was developed with Meteor which is a framework designed for integrating MongoDB with a web user interface. The system was deployed on a server reachable within VTT's intranet and it supported uploads, downloads and basic searches in a web portal. The database instance could be connected to also by other means, such as custom Python scripts. The system was not developed enough to be in actual use, but it did demonstrate a working concept for centralized data storage which would fulfill the needs of the Integrated Photonics group.

Keywords: database system, silicon photonics, MongoDB, Meteor

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Tapio Koukkari

Työn nimi: Tietokantajärjestelmän kehitys piifotoniikan tutkimukseen

Päivämäärä: 15.11.2017 Kieli: Englanti Sivumäärä: 6+47

Mikro- ja nanotieteiden laitos

Professuuri: Fotoniikka

Työn valvoja: Prof. Zhipei Sun

Työn ohjaaja: TkT Timo Aalto

Piifotoniikka on nopeasti kehittyvä tutkimusala. Sitä hyödynnetään monissa sovelluksissa kuten esimerkiksi datakeskusten liitännöissä. Tällä hetkellä VTT:n integroidun fotoniikan tutkimusryhmä tuottaa suuria määriä tutkimustietoa piifotoniikkaan kohdistuvan enenevän kiinnostuksen vuoksi. Tässä diplomityössä suunniteltiin ja kehitettiin tietokantajärjestelmä suurenevan tietomäärän parempaa hallintaa varten. Diplomityössä käytetty tietokanta oli MongoDB, joka on ilmainen ja vapaan lähdekoodin alustariippumaton dokumenttitietokantaohjelmisto. MongoDB valittiin pääasiallisesti sen joustavuuden ansiosta, jota pidettiin hyödyllisenä ominaisuutena nopeasti kehittyvällä tutkimusalalla. Järjestelmän sovellustaso kehitettiin Meteor-arkkitehtuurilla, joka on suunniteltu toteutuksia varten, joissa yhdistetään MongoDB web-käyttöliittymän kanssa. Järjestelmä asennettiin palvelimelle johon sai yhteyden VTT:n sisäverkon sisällä, ja se tuki tiedostonsiirtoa kahteen suuntaan sekä yksinkertaisia hakuja verkkokäyttöliittymässä. Tietokantaan sai yhteyden myös muilla keinoin, kuten Pythonilla ohjelmoiduilla koodinpätkillä. Järjestelmä ei tullut yleiseen käyttöön, mutta se oli toimiva konseptitason toteutus keskitetystä tavasta säilyttää tietoa, mikä täyttäisi integroidun fotoniikan ryhmän tarpeet.

Avainsanat: tietokantajärjestelmä, piifotoniikka, MongoDB, Meteor

Acknowledgements

This thesis was made at VTT Oy, Espoo in Integrated Photonics group. I appreciate being offered a topic that challenged me to seek and apply knowledge outside my comfort zone. I'm thankful for my thesis advisor Timo Aalto, as well as Ari Hokkanen and others at VTT for guidance and support.

I wish to thank the thesis supervisor professor Zhipei Sun for helpful feedback, and help with practicalities. I'm also grateful for the continuous support over the years from my family and friends.

> Try to learn something about everything and everything about something.
>
> *Thomas Henry Huxley*

Helsinki, 15.11.2017

Tapio Koukkari

Contents

Abbreviations

ACID	atomicity, consistency, isolation, durability
AMZI	asymmetric Mach-Zehnder interferometer
API	application programming interface
AWG	arrayed waveguide grating
BASE	basically available, soft state, eventual consistency
BOX	buried oxide
BSON	binary JSON
CAP	consistency, availability, partition tolerance
CMOS	complementary metal-oxide-semiconductor
DBMS	database management system
FSR	free spectral range
FWHM	full-width half max
ISBN	international standard book number
JSON	JavaScript object notation
MZI	Mach-Zehnder interferometer
NoSQL	not only SQL
OEIC	optoelectronic integrated circuit
OODB	object-oriented database
ORDB	object-relational database
PIC	photonic integrated circuit
RDBMS	relational database management system
SLED	superluminescent diode
SM	single-mode
SOI	silicon-on-insulator
SQL	structured query language
TE	transverse electric
TIR	total internal reflection
TM	transverse magnetic
UI	user interface

1 Introduction

This thesis describes initial development stages of a database management system (DBMS) in a research environment. The DBMS was designed for VTT's Photonics Integration research group. The group's work in the Micronova research facility involves utilization of cleanroom environment. Notefully the fabrication of photonic integrated circuits (PICs) made on silicon uses many same cleanroom manufacturing processes as silicon electronics.

The research area has seen much growth in the latest years as it has begun to attract industrial interest in addition to just academical research. The growth has caused increase in the amount of digital information the research group has to store and handle. Currently it is more cumbersome than it could be to share information between project leaders, component designers, process and packaging personnel and people making measurements. Information is currently stored decentralized in networked file systems which make searching for relevant information relatively slow. Finding, for example, many-year-old but relevant measurement results might presently be very time-consuming. With no common and transparent information repository it is also possible to accidentally replicate work without the knowledge of previous efforts. If no solution is presented, the issues most probably will only be magnified in the future with new proceedings in automating many, presently manual, measurement processes. Some examples of the various data items being produced and handled are mask design files for cleanroom lithography, information about the silicon wafers used, measurement instructions and a magnitude of measurement results.

To ease and centralize the handling of research data, this thesis project to conceive a database system for the Photonics Integration group was funded. The envisioned DBMS would be accessible through VTT's intranet, with a possibility of external collaborators gaining limited access at some point. The system was planned to support users friendly, and in some cases semi-automatic, file uploading as well as file downloads in addition to storing possible user-filled metadata fields. In the optimal case any piece of information should have to be entered only once and then be easily found at all relevant process steps. The server that would host the system would be either a physical machine owned by the research group or a VTT-contracted virtual server.

Since the system would be used over a network, it was designed to have a web user interface (UI) that should be easy to use whether in office, cleanroom or laboratory. The UI would be accessed with a web browser. In a relatively fast-changing research environment the data structures in the database should be flexible. Largely for this reason the modern MongoDB was chosen as the system's actual database. The application was written in a JavaScript development framework Meteor which is designed for development of web apps utilizing MongoDB.

During the project it became clear that developing a full-fledged DBMS was out of reach in the project timespan. The project goal was redefined to build a proof-of-concept version of the system that could demonstrate the principal mechanics of how the DBMS would work. The proof-of-concept work was designed to be easily

extendable to allow possible further development.

2 Silicon Photonics

This section covers the basics of silicon photonics as a scientific field. The current interest in silicon photonics research is explained. Some technologies it enables are described, chosen mainly from the focus areas of the author's research group.

Photonics, in contrast to electrons in electronics, uses light (photons) to transmit and manipulate information signals. Vivien and Pavesi mentioned that electronics has for a long time obeyed and benefitted from Moore's law, meaning that the amount of transistors per chip has doubled roughly every 18-19 months by virtue of mass integration [1]. In their accord, integrated photonics hasn't, however, historically enjoyed scale miniaturization at a such a pace. This is in their report due to integration in photonics being more difficult to improve due to a lack of single standard material and process, whereas the basic building blocks in electronics can be manufactured repetitively using the same material and production process. Silicon photonics aims to alleviate this problem by taking the key material of electronics - silicon - and applying it to photonics. Using silicon allows for example utilization of the basic electronics complementary metal-oxide semiconductor (CMOS) manufacturing process [1]. Silicon is transparent at commonly used near-infrared telecommunication wavelengths which is the main reason for its usefulness from a photonics perspective [2].

Presently some additional materials are used alongside silicon with germanium and III-IV semiconductors being the most prominent examples, and silicon photonic devices are growing more complex [1]. Manufacturing is no longer done by a pure CMOS process but CMOS compatibility is still preserved. Commercial interest in silicon photonics is also increasing. Inniss and Rubenstein list a host of companies as investing in and developing silicon photonic technologies as of 2016, including technology giants such as IBM, Intel, Cisco Systems and Oracle [3]. They believed that the tipping point of silicon photonics' adoption in mass market products will be evident at latest in 2021, and cite some silicon photonics researchers who believe the technology is already ripe for mass commercialization.

In the short term, silicon photonics research is driven mainly by optical transport solutions for telecommunications and interconnect technology in data centers which already use some commercialized silicon photonics products [3]. Optical signals in general have much better energy efficiency than electric signals, leading to electric interconnects being replaced by optical ones at ever smaller distances [4]. Inniss and Rubenstein pointed out that silicon photonics is not the only optical technology offering solutions in interconnects, others include e.g. already established indium phosphide and gallium arsenide based technologies [3]. Vivien and Pavesi report that developing silicon photonics can also lead to its utilization in a diverse set of other fields, including e.g. information processing, metrology, sensing and medicine [1]. For example, VTT has done research in sensing and medicine by developing a novel concept for an integrated photonic gas sensor (Figure 1) and developing parts of an optical tomography medical device [4]. The former was realized as part of the EU-funded project MIREGAS [5], and the latter was made in collaboration with a medical company Medlumics within the likewise EU-funded project BiopsyPen [6].

Figure 1: A photonic integrated chip for spectroscopic gas sensing. The chip was manufactured at VTT as part of the MIREGAS project [5]. It utilizes e.g. so-called Echelle gratings, which look like yellow grooves in the image. The chip's dimensions are 5 mm and 10 mm.

Optical tomography is a technique that can be used in the diagnosis of many skin diseases.

2.1 Silicon Waveguides

Waveguides are the photonic counterpart to wires in electronic chips, as their basic function is to simply transmit information on the chip. Integrated silicon waveguides are generally divided into two groups based on their scale. Bogaerts and Selvaraja defined waveguides with dimensions measured in (hundreds of) nanometers as nanophotonic waveguides or photonic nanowires [8]. According to them a typical nanowire has width and height in the order of 300-400 nm. At a larger scale the so-called large-core waveguides are fabricated with dimensions in the order of 1 to 5 micrometers [8]. Both silicon nanophotonic and large-core waveguides are fabricated on silicon-on-insulator (SOI) wafers, which have a buried layer of silica (SiO_2), often simply called buried oxide (BOX), under a thin silicon layer. The high contrast between the optical refractive indexes of silicon and silica help with confining the light signal in the waveguide [8]. A cross-section illustration of a superluminescent diode (SLED) in Figure 2 shows how waveguides are lying on top of the BOX layer. Bogaerts and Selvaraja mentioned that large-core waveguide technology "has

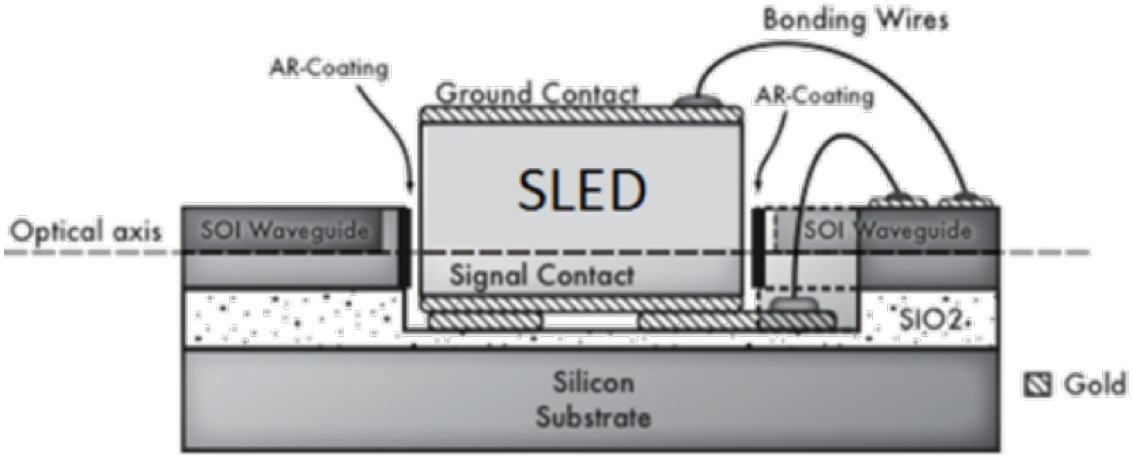


Figure 2: A schematic illustration of a SLED on a SOI wafer [7].

found active use in simple photonic circuits and individual components", but that in contrast to photonic nanowires its "integration density is rather small" [8]. However, the Integrated Photonics group at VTT focuses on large-core technology and its advantages in certain applications are covered later in this chapter.

Fabrication of silicon waveguides has unique challenges regarding power loss as bends and crossings present additional sources of attenuation [2]. Bends cause power loss inversely related to their bending radius which directly affects the achievable scaling of integration. Photonic nanowires don't suffer from bend losses as much as large-core waveguides, and they can manage negligible sub 0.01 dB losses per 90° turns with a bending radius of a few micrometers [9]. In large-core waveguides using simple bends would lead to either impractically large bending radii or unacceptable attenuation levels and for example VTT has thus developed ways to circumvent the issue as will be discussed later. Waveguide crossings cause extra attenuation as well and cross-talk between waveguides that can cause performance problems. Complicated designs for bends and crossings alike have been developed to diminish losses [10]. Subbaraman et al. mentioned that for example a crossing design utilizing an optical phenomenom called Bloch-waves and sub-wavelength nanostructures has shown to achieve a loss of 0.019 dB and crosstalk dampening greater than 40 dB [10].

The research on silicon photonics at VTT is mainly focused on the large-core scale (more specifically called 3 μm scale) [11]. Aalto et al. pointed out that large-core "rib" waveguides have desirable attributes in low propagation loss and optical criteria such as small polarization dependency and single-mode (SM) operation [11]. They say the properties of micron-scale rib waveguides make them especially suited for spectroscopy and signal multiplexing. The thin photonic nanowires work well with only a single polarization and don't have as wide optical bandwidth [9]. Also, due to the size difference of a nanowire and a standard optical fibre, coupling light into a nanophotonic device is inefficient or has to be done through a specific interface component [12][13]. On the other hand, using large-core rib waveguides present a problem in that they require large bending radii which would lead to inconveniently large chip sizes [11]. Two solutions for the issue that VTT has been developing are

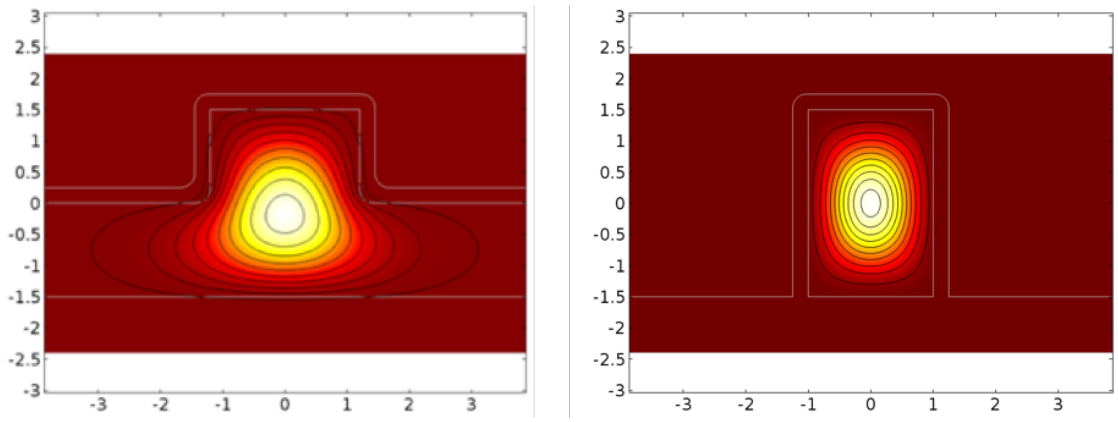


Figure 3: Simulated cross-sectional light distributions in a rib (left) and a strip (right) waveguide [11]. The units are in micrometers.

the so-called "Euler bends" and total internal reflection (TIR) mirrors. The Euler bend design utilizes another waveguide type, strip waveguide, at turning points and has been experimentally shown to enable very small bending radii with feasible loss levels (e.g. 1.27 μm radius with 0.09 and 0.17 dB losses per 180° bend for transverse electric (TE) and transverse magnetic (TM) polarizations respectively) [14]. Rib and strip waveguide structures are illustrated in Figure 3 and Euler bends are featured in Figure 4 and Figure 5.

Another way to produce sharp turns at micron scale is with the aforementioned TIR mirrors. The mirrors rely on total internal reflection which is an optical condition for example commonly utilized in confining light in regular optical fibres. VTT's TIR mirror research has had results in e.g. an elliptic TIR mirror design with a lower insertion loss than its straight-facet equivalent [9]. TIR mirrors are mostly used in

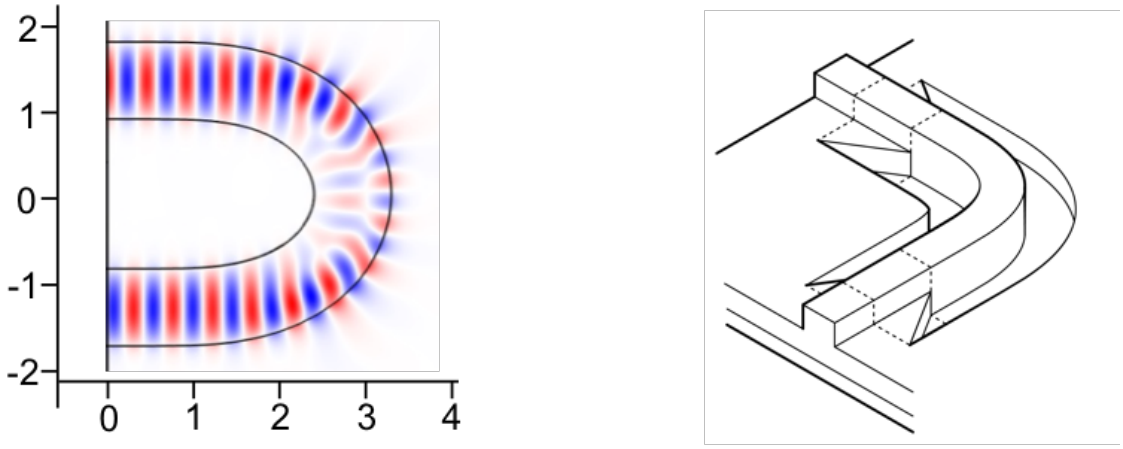


Figure 4: Simulated field distribution in an Euler bend (left) and an illustration of the Euler bend structure (right) with the conversion between rib and strip waveguides visible [14]. The units are in micrometers.

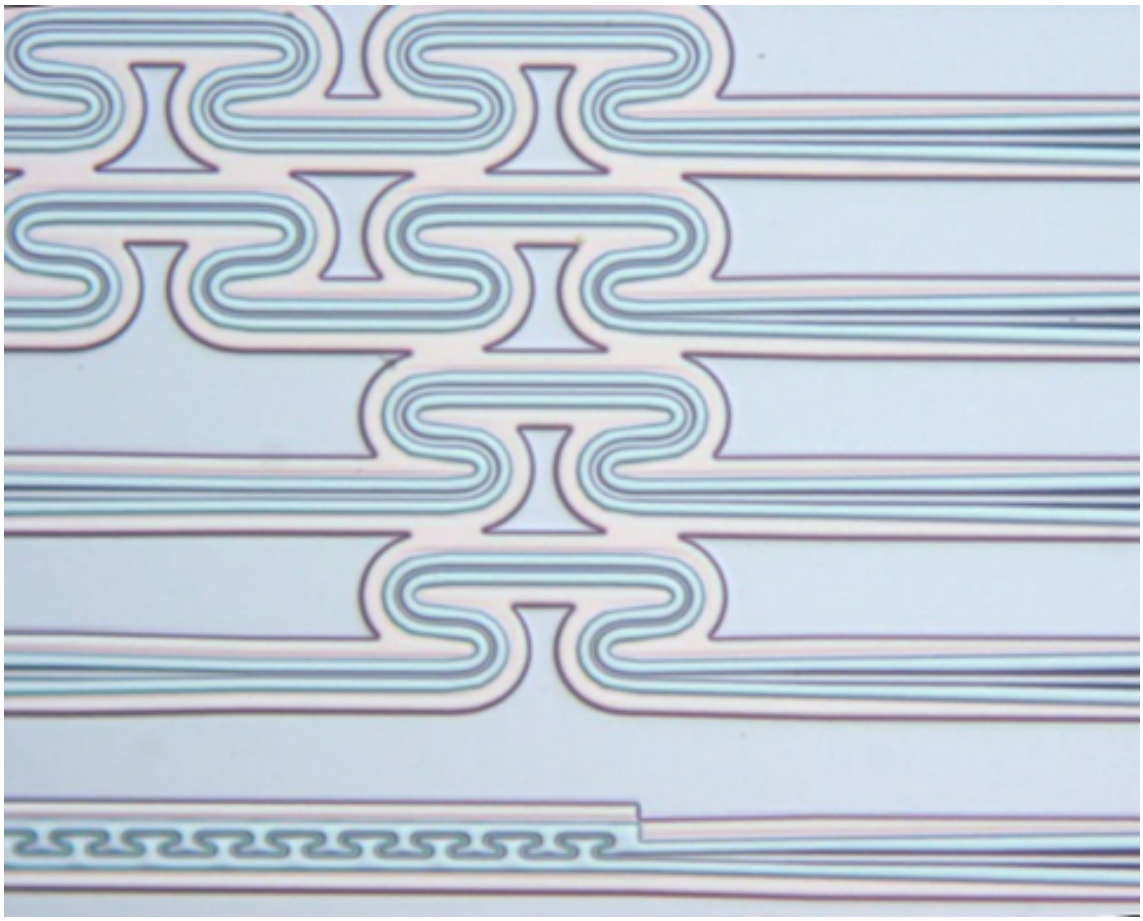


Figure 5: VTT-made Euler bend test structures on a chip as seen with a microscope [7]. The waveguides are spaced at a 50 μm intervals.

large-core waveguides, since in sub-micron nanowires propagating light isn't confined inside the waveguide as much. This could lead to higher insertion losses in the TIR mirrors [4], and, like mentioned earlier, nanowires generally achieve very low-loss bends natively and the needs for nanowire TIR mirrors are low. Larger degree of light confinement is also a reason why low-loss waveguide crossings are easier to realize at 3 μm, although it's possible to compensate crossing losses of nanowires to 0.15 dB with a structure called shallow-etch step [15]. According to Aalto et al., TIR mirrors at micron scale can achieve as low as 0.1 dB losses for both rib and strip waveguides [11]. TIR mirrors used in 90° turns are shown in Figure 6.

2.2 Optical Waveguide Components

This section describes how optical phenomena are utilized in some common optical waveguide components that can be used in pure optical operation and purely photonic chips. Since they are electrically "passive", these non-electric optical components are often called passive components. Many passive component designs can have electro-optic variations, however, (turning them into "active" components) and they

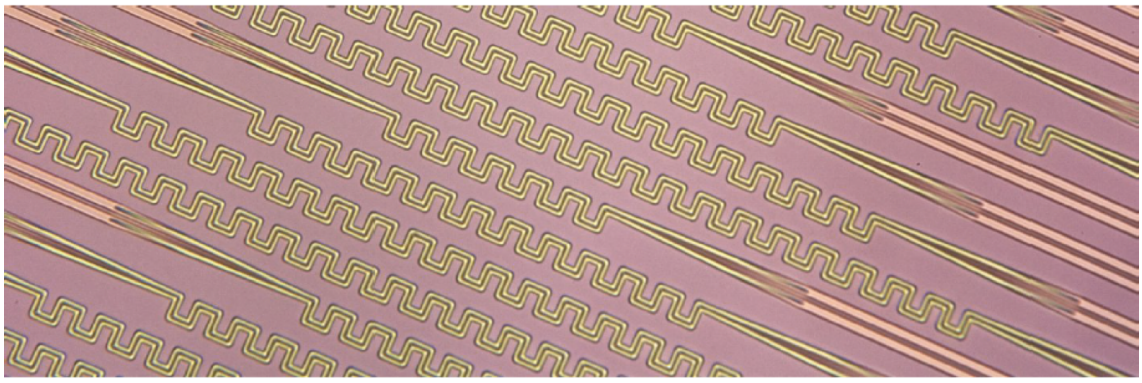


Figure 6: A microscope image of VTT-made TIR mirror test structures [9]. The waveguides are spaced at a 50 μm interval.

are noted when appropriate.

Ring resonators are closed waveguide loops that can be used in network operation as modulators, filters and switches [2]. Basic operation of ring resonators is illustrated in Figure 7, and an example of coupled resonant light spectrum is shown in Figure 8. Light waves propagating in a passing-by waveguide interact with the rings in their so-called resonant modes. When the waveguide is transmitting light, resonant wavelengths will couple into an adjacently placed ring, while light in other wavelengths will pass it unhindered. The ring resonator can thus be designed so as to capture certain desired wavelengths, and adding a second output waveguide enables a basic switch or filtering function where light is divided into different outputs based on its wavelength. Bergman et al. pointed out that the ring resonator can be also made with an electro-optic design, where the wavelengths it captures can be electrically altered [2]. This means that the electro-optic variant can modulate light and thus

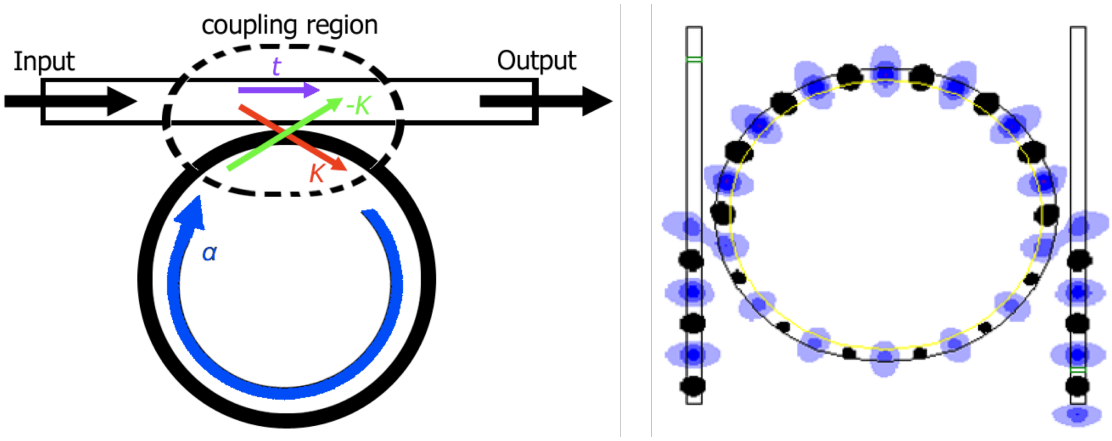


Figure 7: Ring resonator schematic illustrations. K and t in the left image [16] are called the coupling and transmission coefficient respectively and determine which wavelengths are coupled in the ring. The right image [17] shows a ring resonator redirecting input resonant light.

generate optical signal onto a uniform input light. As such it can be used as a core element in transforming an electric signal into an optical one.

Devices called Mach-Zehnder interferometers (MZIs) split light into two waveguide branches which are later joined back into a single output, as seen in Figure 9. When the branches are different in length, the device is called asymmetric (AMZI) and the two light beams experience phase shift in relation to one another [19]. This causes interference between the beams and shifts in the spectrum of the output when compared to the pre-split input light. As was the case with ring modulators, Mach-Zehnder interferometers can contain electronically active elements which allow for electric modulation of the output light [19].

Arrayed waveguide gratings (AWGs) are used in optical multiplexing and demultiplexing, i.e. in combining or reading singular signals into/from a long-range transmission beam. For example, in the demultiplexing of light as illustrated in Figure 10, the AWG is used to separate many signal wavelengths from an input source each to their own output channels [19]. In this case the input light enters a

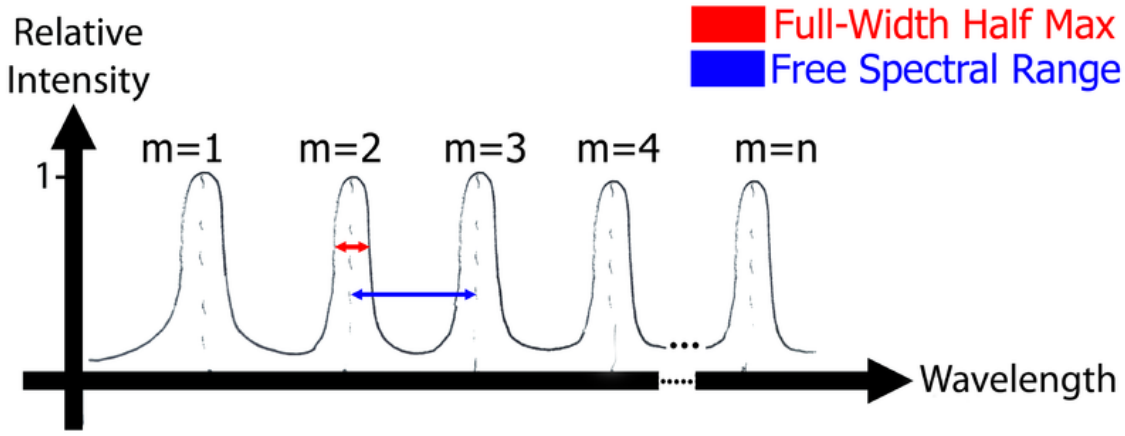


Figure 8: An illustration of a light spectrum containing spikes of resonant wavelengths [18]. A device like a ring resonator, AMZI or AWG can output spectra resembling this. Full-Width Half Max (FWHM) is a parameter describing the wavelength width where intensity is at least half of the peak's maximum value. Free Spectral Range (FSR) is the wavelength separation between peaks.

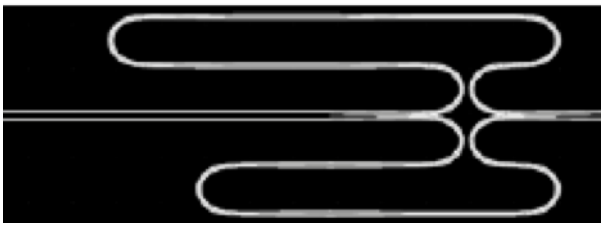


Figure 9: An image depicting an AMZI structure [11]. Input light is divided into two asymmetrical branches which then terminate in the same output. The waveguide bend radii are 50 μm.

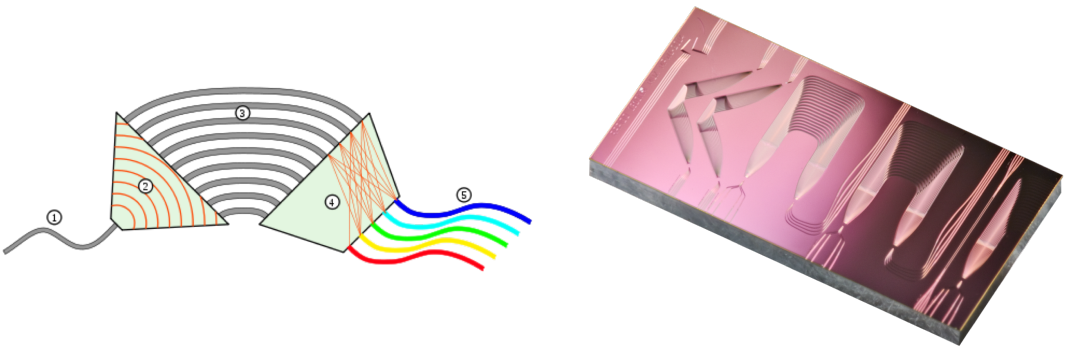


Figure 10: On the left is a schematic of the AWG operation [20]. Light enters from an input (1) into a coupler (2) and is divided into multiple channels (3). The channels feed into another coupler (4). Due to interference all output channels (5) pick only selected wavelengths (illustrated here with different colors, as would happen if the input was white light). A 5x10 mm^2 photonic chip with AWG test components by VTT is seen on the right.

brief free-space (in the optical sense) coupler and feeds into multiple channels. The optical channels differ in length and terminate in another free-space coupler. Much like in the AMZI, the light in the output coupler interferes with itself as a result. When the AWG's output channels are spaced correctly, each captures a specific constructively interfered wavelength and omits other wavelengths that vanish due to destructive interference. Thus the AWG can be used to convert input light containing many signals at different wavelengths into multiple outputs each containing a single signal, i.e. the AWG works as a demultiplexer [19]. Multiplexing is achieved by the reverse operation. Deen and Basu point out that AWGs can be designed also with multiple input channels instead of only one [19].

Diffraction gratings in general optics have historically been used to separate light into its component wavelengths, and as such could be used like AWGs in demultiplexing [19]. Fabrication difficulties in making planar diffraction gratings have held back their adoption in silicon photonics, but that a particular design called Echelle grating (originally presented for optics in 1949 [21]) is a promising exception. The Echelle grating, as seen in Figure 11, consists of multitudes of small "teeth" which form a grooved planar grating. The grating reflects input light which, due to the teeth-induced optical phenomenom diffraction, experiences slight deviation in its angle of reflection depending on its wavelength. Dabos et. al. mentioned that the Echelle grating can be used in many same ways like the AWG and it has some advantages like its smaller planar size and its "robustness in wide number of applications" [22]. VTT-developed Echelle gratings can be seen in the MIREGAS gas sensor in Figure 1.

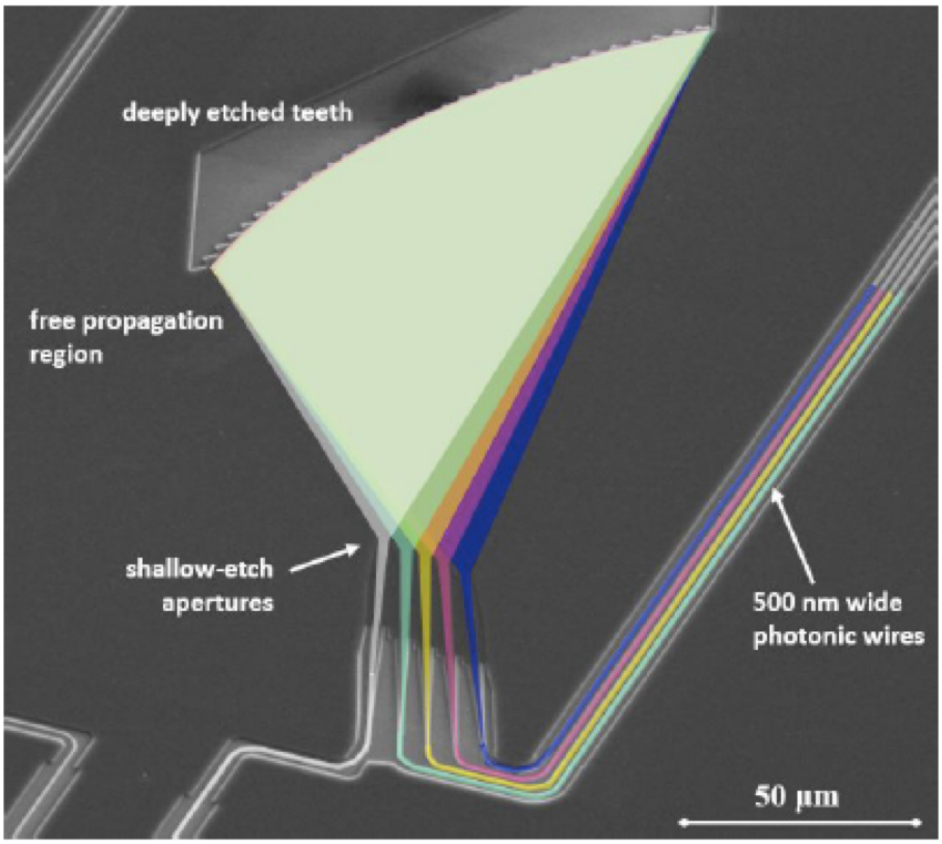


Figure 11: An illustration of an Echelle grating [23]. The leftmost channel is the input channel and the output channels pick different wavelengths of the reflected light.

2.3 Monolithic vs Hybrid Integration

This section explains two principal design implementations of PICs and optoelectronic integrated circuits (OEICs). In the so-called monolithic integration all optical and electronic functions are fabricated on the same substrate material [3]. Hybrid integration on the other hand means combining components made from two or more different materials. Inniss and Rubenstein point out that hybrid integration allows for using the best materials for each function, but that combining the different materials presents "its own design and manufacturing challenges" [3].

Silicon can be used in both hybrid or monolithic integration. However, as noted by Inniss and Rubenstein, silicon cannot lase and thus lasers can't be fabricated on a silicon monolithic chip [3]. As such any silicon based PIC that needs a common laser has to rely on hybrid integration.

To avoid an undesirable external coupling between a silicon photonic chip and a laser, a hybrid technique Inniss and Rubenstein call "heterogeneous integration" can be used to bond the laser onto the silicon chip directly [3]. They say that it can be utilized for many optical features which are rarely integrated in other platforms, such as optical isolators and circulators in addition to the laser. They also point out

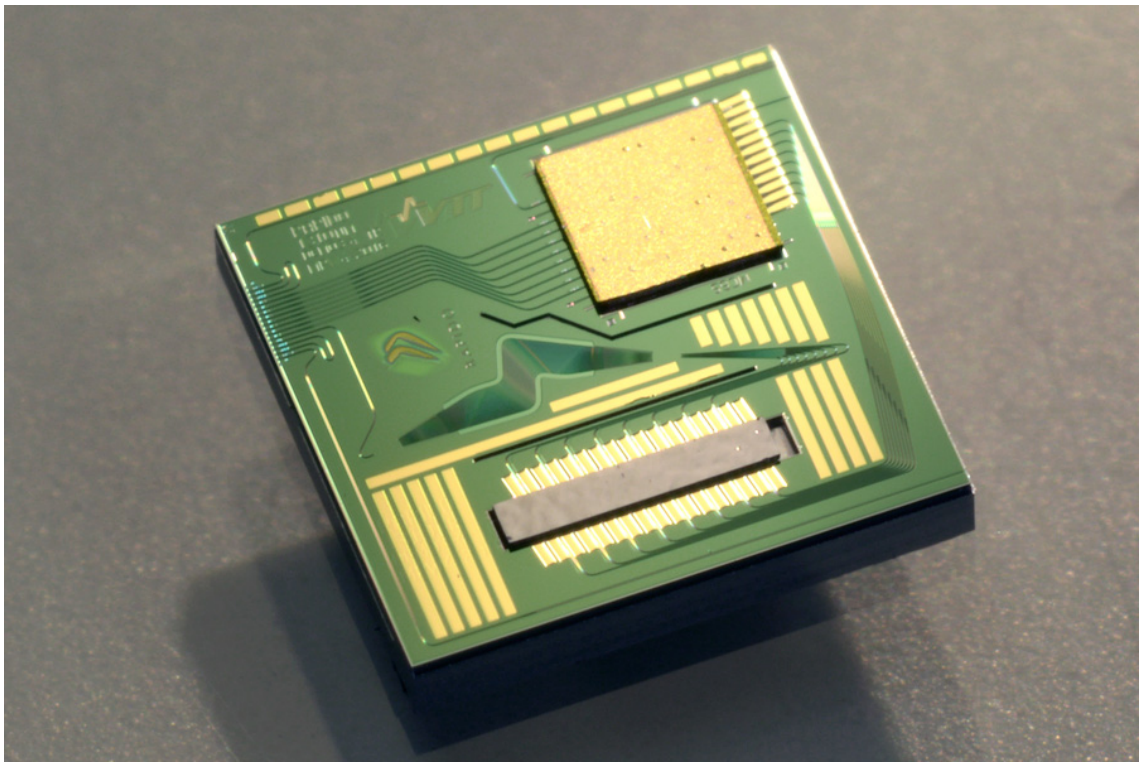


Figure 12: A 5x5 mm^2 PIC developed at VTT as a part of the RAPIDO project. The chip has been made with a hybrid integration approach. Components like the middle AWG have been fabricated onto the chip itself, while others have been bonded onto it afterwards.

that using heterogeneous integration results in a single chip resembling a monolithic silicon photonic chip despite the different original substrate materials [3]. VTT developed hybrid integration and the bonding of components in e.g. the EU-backed RAPIDO-project [24]. Figure 12 shows a PIC fabricated in the project.

2.4 Workflow and Research Data

This section roughly describes the workflow of silicon photonics research and details the sort of data that is produced. Why a centralized databank would be useful is explained. Plans on how the data would be structured in the database are also detailed.

When developing a new component, a designer generally runs many simulations to find suitable parameters. When the concept of the component is ready, the designer creates so-called mask files which resemble layered images detailing different steps of fabrication. The mask files are at minimum chip-level designs which include multitudes of components. Typically a test chip might include multiple variations of a couple of different component types. The chip-level mask files are usually concatenated into reticle masks. Reticles are the smallest non-divisible units when a wafer-level layout is planned. When fabricating test chips it's typical to have a few

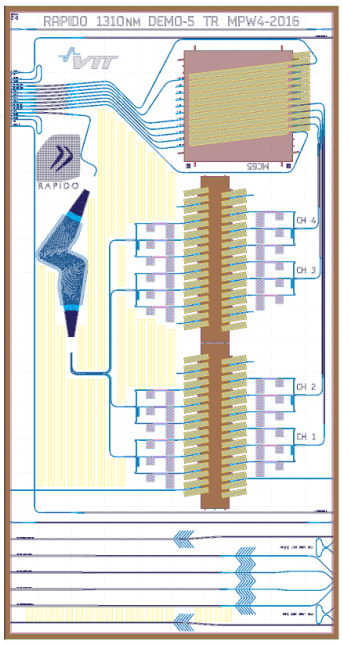


Figure 13: A single chip mask design with dimensions of 5 and 10 mm.

reticles repeated across a whole silicon wafer. An example of a chip-level mask file can be seen in Figure 13 while one reticle is shown in Figure 14.

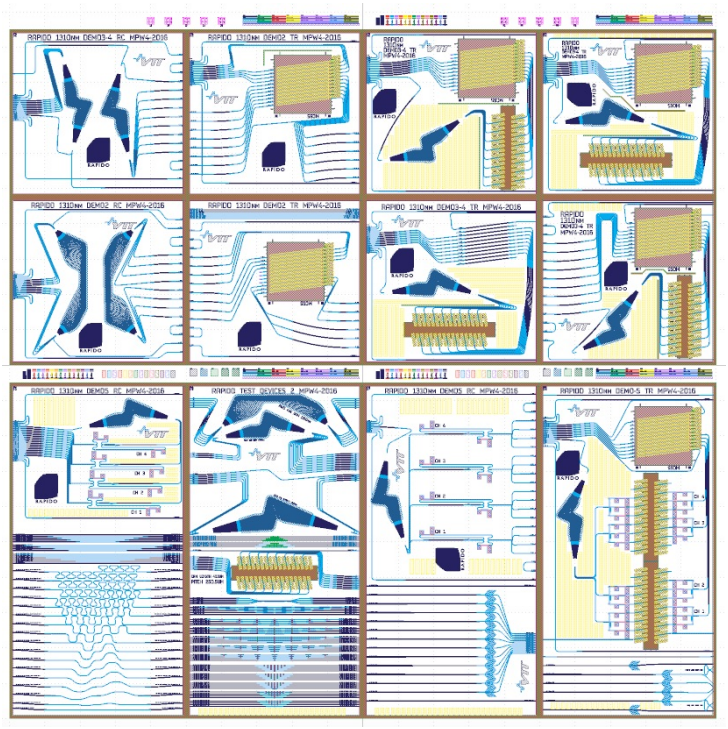


Figure 14: A reticle mask image containing multiple chip designs. The reticle is 20x20 mm^2 in size.

Many physical reticle masks are used in patterning one silicon wafer by a procedure called stepping. In addition, sometimes so-called contact masks can be made. Contact masks are used to pattern a whole wafer simultaneously. They are needed for example to create alignment marks for double side processing and when the chip's length and/or width is more than 20 mm. The component fabrication process has multiple stages and steps depending on the component types, and can include cleanroom processes such as wet and/or dry etching, annealing, epitaxy, bonding etc. The externally provided wafers themselves are usually SOI-wafers and can have different material properties and layer thicknesses. Figure 15 shows a set of wafers in a cleanroom.

After being fabricated the wafers' material properties, like electric resistance, may be tested in the cleanroom. Likewise the components themselves can be measured electrically in the cleanroom even while the wafer is still intact. Finally the chips are cleaved apart from the wafer and, if they contain optical waveguide components, they typically enter a measurement lab where e.g. their optical transmission properties are measured. Figure 16 illustrates the hierarchy between a batch of wafers and a single component.

A work process like the one described above creates lots of new digital information as well as requiring process descriptions and measurement instructions. In component design, simulation results and mask files are created and the cleanroom process and measurement instructions have to be planned. In fabrication, process specifications are needed and great deal of measurement data is also produced. In the optical measurement lab a myriad of transmission tests, like one depicted in Figure 17, and e.g. microscope imaging is done. With the industry seeing rapid growth in recent years, there are currently more people doing much larger volumes of research on silicon photonics at VTT than before.

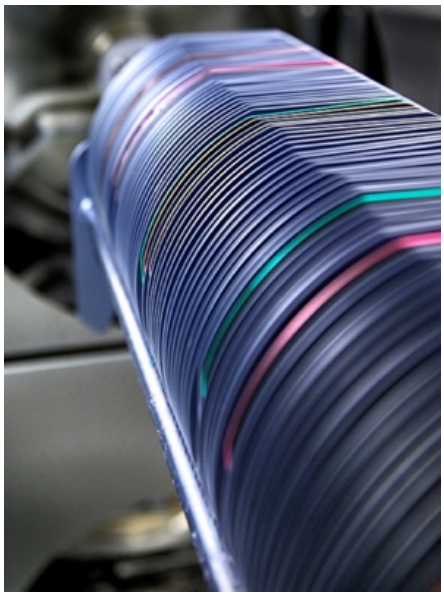


Figure 15: Silicon wafers in cleanroom waiting to be processed.

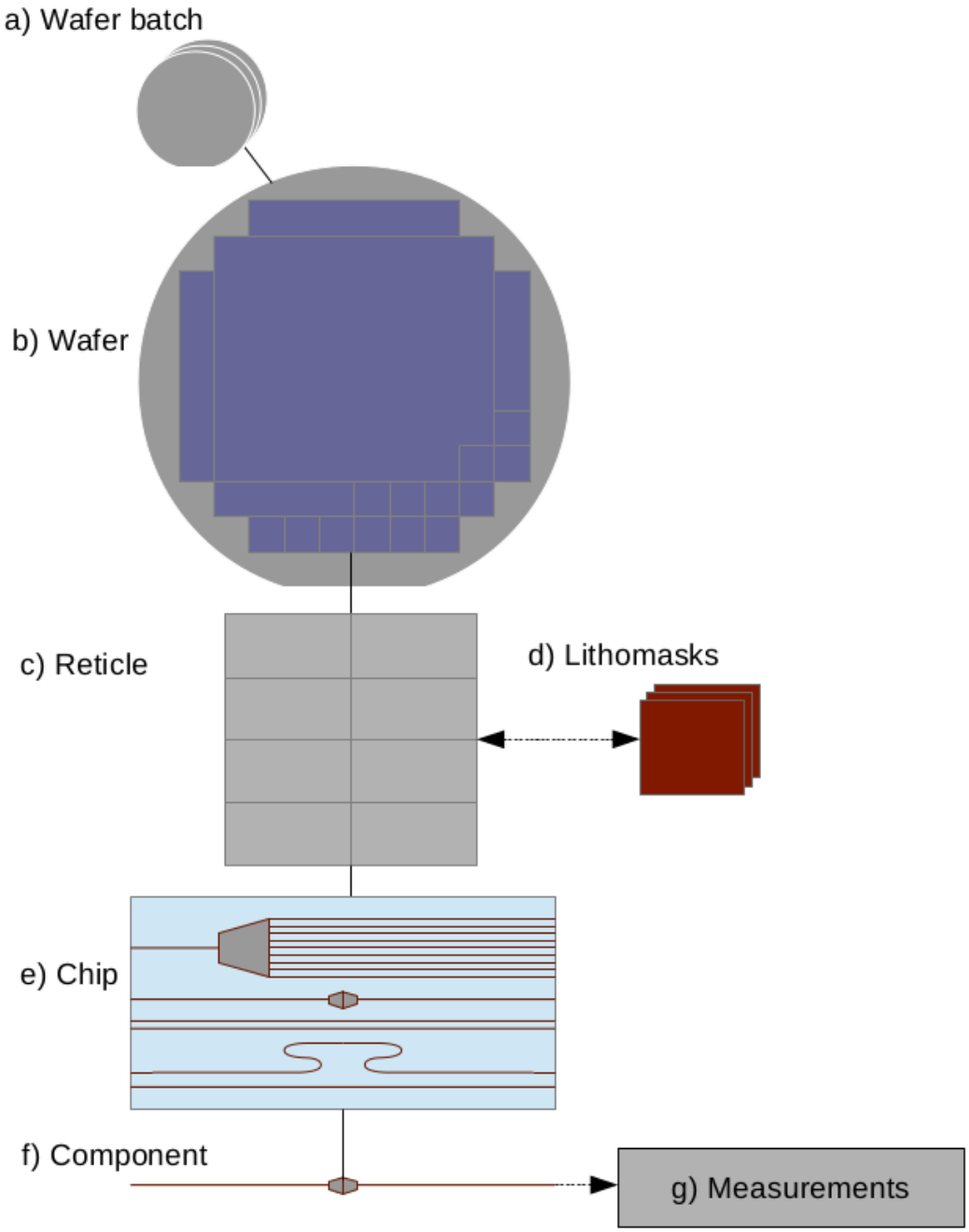


Figure 16: The physical hierarchy relating a measured component to a batch of wafers. Reticles are groups of chips which are repeated multiple times on a wafer (often a single reticle is repeated on a whole wafer). Lithomasks refer to contact and reticle masks which are tools used in lithography process. The physical hierarchy would not necessarily have to be replicated one-to-one in the data structure of the database.

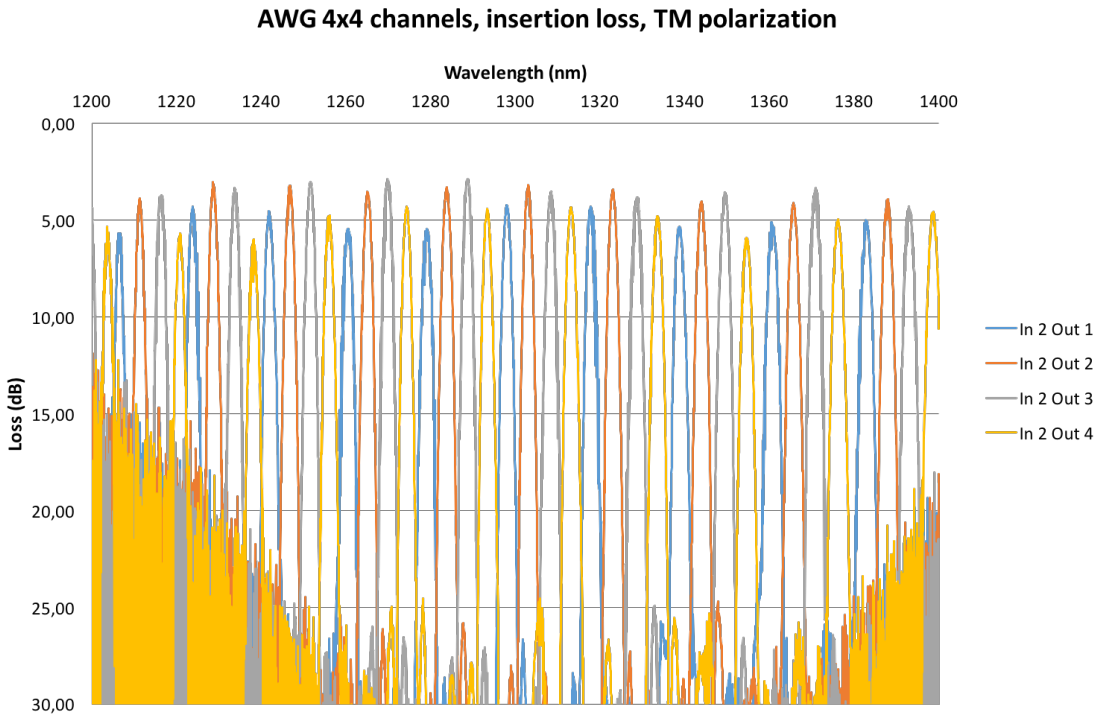


Figure 17: Graph of coupling attenuation results from an AWG (4x4 channels) measurement made at VTT. The graph depicts four attenuation spectra measured by inserting light into an input channel (2) and measuring transmitted light from the different outputs with a spectrum analyzer. The figure demonstrates for example how transmission peaks shift between output channels.

The increasing amount of produced data has problems in its handling. Earlier with fewer people, using file systems was adequate but currently information is increasingly hard to find in sprawling folder trees. This concern leads to the planning of a centralized database system, which was being developed in this thesis work. Figure 18 shows a possible way to structurize data and files in the proposed databank. With everyone using a properly structured shared database system, information should be easy and fast to both save and find. Additionally, a database system could assist in long-term stability monitoring for measurements. For example doing a comparison of results over time, like with the spectra seen in Figure 17, lets researchers to find possible differences of optical properties between chips, wafers and process runs. In a case like this a DBMS would help find relevant measurements fast, and a specialised feature designed for automating such comparative analysis could be integrated at some point.

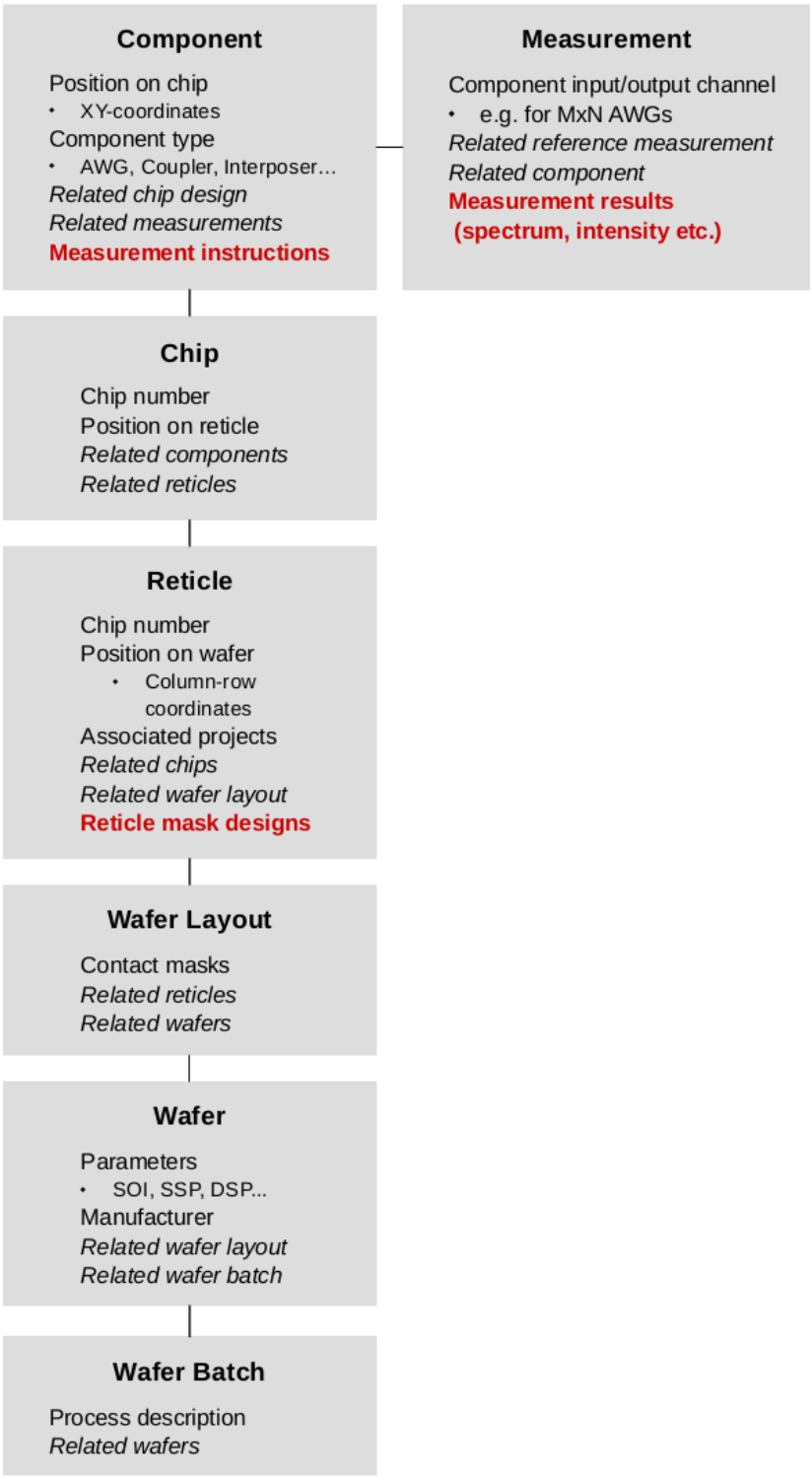


Figure 18: One possible design for the data structure in the finalized database system. Text in italics refers to relations to other illustrated data collections in the figure. Text in red refers to related file storing GridFS collections, which aren't otherwise pictured here. Relations between collections are not necessarily singular, i.e. many Measurement documents can be related to one Component document etc. The GridFS file storage method is discussed in Chapter 3.

3 Survey on Database Design

The goal of this section is to give the reader an overview of different database structures, design approaches and purposes. A brief description on the history of database design is given and then more attention is directed to the approaches relevant to this thesis study.

A database itself is essentially a stockpile of information that can exist for years [25]. The database is usually handled by a specialized software system called a database system or, more formally, a database management system (DBMS). Ullman and Widom list five tasks that a DBMS should be able to perform [25]:

- Users should be able to use a specialized data-definition language to create new databases and their logical structure, i.e. specify their so-called schemas.
- Users should be able to query (ask a question about) the stored data, as well as modify it using a specific language, oftentimes called a query language or data-manipulation language.
- The system should support the storing of enormous data volumes up to terabytes and more for long periods of time, as well as efficient accessibility to the data.
- The system should be durable and have failsafes so that the stored data can be recovered when errors, failures or misuse are encountered.
- The system should provide access to many users simultaneously so that their actions are isolated from each other and can not cause unexpected interactions. User made actions should be atomic, as in they should always be fully completed or alternatively fully disregarded in the case of a failure. No partial actions should be able to be committed on the system.

File systems that preceeded database management systems lacked many of the criteria listed above, and the first DBMSs were developed in the 1960s to serve mainly applications which handled many small and frequently accessed data items. [25]. Applications of this type include banking systems, airline reservation systems and corporate record keeping. Most prominent structure models in the early DBMSs were the tree-like hierarchical model and the graph-based network model. These early models lacked support for high-level query languages [25].

In 1970 an article called "A Relational Model of Data for Large Shared Data Banks" by Ted Codd from IBM Research Laboratory proposed a new database model, which was based on relational mathematics theory [26]. By the 1990s the relational database management system (RDBMS) was used as the norm [25]. At this point object-oriented programming languages like C++ were gaining popularity but their adoption in DBMSs lead to a problem, impedance mismatch, especially in intensive data handling applications [27]. The impedance mismatch refers to the different approaches to data structure in RDBMSs and object-oriented languages, which causes translation and conversion issues in interfacing the database layer with the application layer. As a result object-oriented databases (OODBs), and later object-relational databases (ORDBs) were developed [27].

In the early 2000s the growth of web-scale data handling led to development of new database technologies, collectively named NoSQL [28]. The Structured Query Language (SQL) is the de-facto standard query language used in RDBMSs [29], and NoSQL literally refers to "no SQL" or in a later interpretation "not only SQL" [30]. Vaish pointed out that no matter its root phrase, the key concept in NoSQL technologies is the non-adherence to RDBMS design principles [28]. In present day NoSQL is used as an umbrella term covering all non-relational database systems that were mostly created with a distributed and parallel computing environment in mind [30]. NoSQL approaches include, but are not limited to, column-oriented, key/value pair, document, and graph database designs [30].

The relevant and historically significant relational approach is described next in more detail. The NoSQL document store archetype, which uses a semi-structured data model, is given attention afterwards. Finally there is an overview on different web applications which outlines reasoning behind the technologies chosen for the DBMS developed in this thesis project.

3.1 Relational Approach

Relational database theory is heavily based on relational algebra and calculus. The mathematics have been described extensively by Elmasri and Navaste [31]. In practice basic RDBMS concepts are relatively simple to understand without deep knowledge of the theoretical basis.

Relational databases store their records in collections resembling common data tables [32]. One stored data item is equal to one row on a table. A table's columns enforce the metadata fields a record will have [33]. Columns have two important characteristics, name and domain. The name describes what will be stored under the column, and the domain dictates what data types it accepts [33]. Data types include but are not limited to numbers, pieces of text and timestamps. For example, a simple "bookcase" table might have columns for "title", "author", "edition" and "publication date" with the first two columns accepting only text and the two latter an integer number and a timestamp respectively. All data rows have to include all columns in one way or another and thus even a data field with no known value must be created. In a scenario like this a special 'null' value type can be used to indicate "empty" fields [33]. Churcher pointed out that the use of null values can be allowed or prohibited per-column and as such in some cases a row has to include an actual value for certain columns [33].

To ensure all data records are distinguishable from one another, the fields in one column or a combination of columns are used as so-called primary keys [33]. Primary keys should always be unique so they can be used to find exact data records. In the bookcase example, no single column would work as a good primary key column since none of them hold fields that are guaranteed different from one another. A combination of columns might provide better options, but the easiest way to implement proper primary keys would probably be adding a new column for the books' International Standard Book Numbers (ISBN).

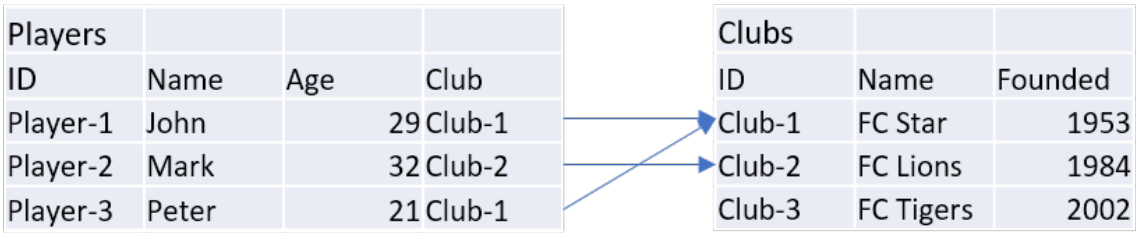
Primary keys are also used to form relations between data records. A record in

one table can refer to a record in another table by storing the other record's primary key. The referring key value is called a foreign key [33]. For example, if one table stored football players and a second table football clubs, a player row could store a club's primary key as a foreign key to link the player to their club. An example of table structure like this is shown in Figure 19. So-called self relationships are also possible with foreign keys, which means that the referred record is stored in the same table as the record making the reference [33].

The structure of the database, i.e. the table definitions and all relationships between them, is called a relational database schema or just a database schema. Relational database transactions should fulfill criteria called atomicity, consistency, isolation and durability (ACID) [25]:

- **Atomicity** - The transaction should be undivisible, i.e. it should be either fully processed or fail completely.
- **Consistency** - Refers to the logical boundaries of the database which the transactions shouldn't be able to cross. For example, in an accounting database cash balances shouldn't probably be able to go negative and as such consistent transactions shouldn't be able to make them so.
- **Isolation** - Transactions should be executed without interfering with each other, i.e. no transaction should care about other transactions while being processed.
- **Durability** - A condition that the effect of a transaction, after being processed, should not be able to be lost (unless reversed by another transaction).

As amounts of data grow and especially if there are many columns in a table, proper indexing is paramount to efficient querying. An index is essentially a parallel structure to the table that collects values from a column or few and uses them to uphold an ordered set of row pointers. Churcher explained database indexes being analogous to terminology indexes in backs of books, which similarly enhance finding relevant information [33]. Using indexes can make large differences in query speeds but inserting and removing rows requires updating the table's indexes as well, which means that underused indexes should be avoided. Churcher pointed out that since



Players			
ID	Name	Age	Club
Player-1	John	29	Club-1
Player-2	Mark	32	Club-2
Player-3	Peter	21	Club-1

Clubs		
ID	Name	Founded
Club-1	FC Star	1953
Club-2	FC Lions	1984
Club-3	FC Tigers	2002

Figure 19: A table structure showing how a relation could be made between two tables. ID-columns are presumed to be unique per row and used as the primary keys. Primary keys are used to make relations to other rows.

adding indexes is a tradeoff between rapid querying and update speed, the type of the application should be taken into account when planning if and how many indexes should be used [33].

Handling of standard relational database systems happens with the query language SQL [29]. Darmawikarta recounted that SQL was developed at IBM in the 1970s and if was formalized in 1986 by American National Standards Institute, after which it has been revised every couple of years [32]. He noted that SQL is used to both define data structure (create, delete and alter tables) and manipulate data itself (insert, retrieve and update records stored in the tables). Elmasri and Navaste pointed out that later iterations of SQL have borrowed ideas from object oriented database designs, and many commercial databases using SQL can thus be called object-relational [31].

3.2 Document Stores

The document store, or document-oriented database, is an important NoSQL database archetype. Despite the name, document stores are not content management systems for documents in the traditional sense like Microsoft Office, but instead store data in sets of loose key/value pairs which are referred to as documents [34]. While documents are roughly analoguous to rows in relational tables, the database considers the documents as singular objects. Collections, which somewhat resemble tables in RDBMSs, can therefore contain documents with differing structures instead of enforcing a strict schema [34]. Still, as Tiwari pointed out, some shared structure between documents is advisable to achieve efficient querying of the database [30]. Changlin noted that documents are usually stored in a format called JavaScript Object Notation (JSON) [35]. An example JSON document is shown in Figure 20, which also demonstrates so-called embedded documents in values for address and the list of phone numbers. The document store MongoDB, important to this thesis, uses a binary JSON format (BSON) which from a data-handling perspective changes little but adds some additional features and optimization [37].

One of the theoretical foundations of NoSQL databases, according to Changlin, is the consistency-availability-partition tolerance theorem (CAP) [35]. Tiwari explained the three conditions as follows [30]:

- **Consistency** - Each client has same view of the data.
- **Availability** - Each client can always read and write.
- **Partition tolerance** - System works well across distributed physical networks.

CAP theorem states that at a time any combination of two of the CAP conditions can be fulfilled, but all three cannot be achieved simultaneously. For example, Changlin noted that relational databases by default are consistent and highly available, but as a result they suffer from bad partition tolerance and thus perform poorly when distributed across multiple servers over a network [35]. According to Tiwari, a distributed RDBMS usually chooses to sacrifice availability to achieve the necessary

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```


Figure 20: An example JSON document depicting a person [36]. Quoted green text indicates fields or keys. Values can be of different types, e.g. text strings (quoted red text), special types (null, true) or lists or embedded documents where there are associated fields under a "parent" field.

network functionality [30]. Document stores, along with other NoSQL databases, are also bound by the CAP theorem but might balance the conditions in different ways. An important distinction is that instead of the strict ACID principles of relational databases, NoSQL solutions use an approach based on principles called basically available-soft state-eventual consistency (BASE) [34]:

- **Basically available** - The database will return a response, but that response might be an error or in an inconsistent state.
- **Soft state** - The data in a distributed database can change over time as it seeks internal consistency.
- **Eventual consistency** - Given time, a change in data will become consistent across a distributed database.

In a practical application, Changlin pointed out that BASE is a compromise that prioritizes accomplished data reads by allowing read-write separation (writes can be delayed without serious impact) and new-old data separation (reads may return slightly out-of-date responses) [35]. One of the advantages of NoSQL databases is that to meet growing workloads more servers can be added into a distributed database system, whereas RDBMSs mostly have to increase the computing and memory capabilities of already existing servers [34]. In other words relational databases scale vertically while NoSQL databases scale both vertically and horizontally. On the other hand, as Tiwari put it, "ironically, one of the features most missed in NoSQL is SQL", referring to lesser power in transactional integrity and query flexibility [30].

Document stores handle relations similarly to the RDBMS foreign key concept, as pointed out by Mason, by storing pointers to other documents in any collection in the database [34]. Also similar to relational databases, document stores support the use of indexes to speed up query operations. The importance of indexes goes back to why a "soft schema" is useful in document store collections, even if the schema can be altered over time without redefining the whole collection. Important in the context of this thesis, due to the aforementioned features the planning of a data structure in a document-oriented model is made similar to one in the relational model, but flexibility to handle later alterations is increased.

One aspect of modern database systems is how to handle arbitrary and large data objects like video clips and pictures, which in regards to this work was also of interest due to the amount of e.g. pdf-files wished to be stored. Hows claimed that the solutions offered by traditional RDBMSs like MySQL and PostgreSQL are "bolt-ons" and as such not always easy to use [37]. He continued that MongoDB, while limiting its document size to maximum of 16 Megabytes, natively supports a file system specification called GridFS. Storing files in MongoDB splits them between two collections, where one holds metadata like filename and size, and the other further divides the file content and stores it in data chunks, each of which is a document in the chunks collection [37]. Reading a file correctly from the database automatically "rebuilds" the file without user complications.

3.3 Database System Applications

How do RDBMSs and NoSQL approaches perform in different applications when compared to each other? In discussing this question, Vaish listed 3 archetypes of database systems: transactional, computational and web-scale applications [28]. Transactional applications are defined, according to Vaish, by reliance on consistency and integrity of the data and relatively low usage rate of the database. The application talks to one database instance at a time and no replication or load-balancing is needed. A transactional application usually needs a highly structured schema with defined relationships. Because of these requirements a traditional relational database structure is a good choice. While some NoSQL databases support ACID and transactional operations, Vaish said they would require more effort for no substantial gain [28].

A computational application, as defined by Vaish, self describedly processes a

lot of computation in the application layer [28]. Computational applications usually don't need as much relations between data entries as transactional applications do. Data should become consistent relatively fast, but small latency is acceptable. Data retrieval is usually more vertical than horizontal, i.e. often the application needs few data values from many objects instead of many values from a single object. With these general requirements, both RDBMS and NoSQL system models have advantages and disadvantages. RDBMSs have better consistency and can enforce a rigid schema and handle relationships in the database layer. Document store NoSQL databases can uphold a schema but they can't enforce it. Formal relations have to be maintained in the application layer if using a NoSQL database. However, Vaish noted that NoSQL systems are often optimized for partial data retrieval which serves vertical processing better - RDBMSs retrieve whole rows (objects) and apply any given filters afterwards [28]. Because of this in very large scale applications (where amount of records exceeds few hundred millions) NoSQL has a faster performance and is usually the better choice. In mid-to-large scale applications either system can work. At a smaller scale NoSQL might undesirably need more setup and tuning than a RDBMS [28].

The third archetype listed by Vaish is a web-scale application. Web-scale applications have enormous data volumes and have multitudes of users that can connect from across the globe. Consistency is not as highly prioritized and while relations exist the data schemas can change often. Vertical processing performance is preferred and since ACID operation is not a requirement, according to Vaish, NoSQL models are almost universally the better choice [28].

4 Choosing Software Tools

This chapter along with the following ones describe the work done to create a database system for use in VTT silicon photonics research. The early phases of the database project are covered in this section. The initial software requirements are specified and why certain technologies and approaches were adopted are researched. In discussing options, it is understood that a large amount of software tools are named in rapid succession which may be somewhat confusing for the reader. For this reason the tools relevant to later parts of this thesis are summarized in the chapter's last paragraph.

The motivation for the database system was largely to ease the handling and storing of increasing amounts of research data. In addition to the basic needs of such software some additional requirements for the system were specified:

- The database system should be flexible so that future needs can be accommodated without major overhaul.
- The application's user interface should be a web portal that supports searching in addition to uploads and downloads. Later on the user interface should be user specific, i.e. a component designer should be able to easily insert their designs follow their realization while a lab technician should effortlessly find measurement specifications and expected results.
- The user interface should be relatively easy to understand and use to lower the barrier of entry.
- While at some point collaborators could possibly have limited access to the database, the system should primarily be developed for VTT internal use.

An early pitch for the application involved using a relational database like MySQL as the system's underlying storage solution. An argument for using a relational database would be for example the fact that from the three archetypes discussed in the previous chapter, the system would resemble a transactional DBMS the most. However, the flexibility provided by NoSQL document stores prompted interest in them. A document store database like MongoDB should offer more structural adaptability than what an RDBMS does since no rigid schema is enforced [30]. For example, two documents in a MongoDB collection don't necessarily have to share any common fields (with the exception of a "MongoID" field). While in practice storing totally different documents in the same collection wouldn't make much sense, the functionality allows adding data fields to new document entries without having to update all old documents in the collection. In an RDBMS if a new metadata field was added all of the table's old rows would have to be updated as well [30]. In a research environment where components and measurement specifications can change relatively frequently, it's highly likely that measurements done in a few years will keep track of more or different variables than ones done today. In addition to the adaptibility of the document store technology the MongoDB document store is noted by Tiwari to be widely used and supported in present day web applications [30]. As such MongoDB was chosen for the database layer of the system.

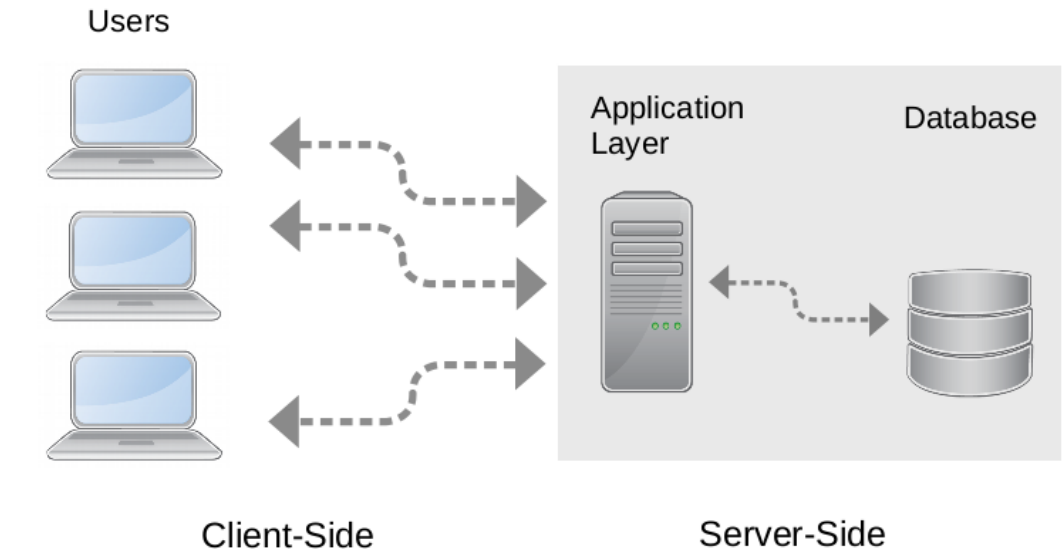


Figure 21: A basic overview of how using the database system would work. Users ("clients") connect over a network to a server that hosts the web application and the database just as any web page on the Internet. The user clients' web browsers load the web portal as supplied by the server. Besides providing the web portal, the server's application layer processes and approves requests received from users and acts as an interface for the database.

Since the application was going to use a web user interface (UI) layer the selection of a suitable framework stack was an important decision. Frameworks in web development offer ready-made structure and package libraries that developers can utilize so that they don't have to design and program everything themselves [38]. The Google-developed AngularJS (or just Angular) seemed like a prominent front-end framework for dynamic web pages. Notably it was also discovered that Angular is used in the so-called MEAN-stack (Mongo-Express-Angular-Node) for web apps using MongoDB in their back-end [39]. Express is another framework and Node is a technology powering Angular and Express and many other programming tools. Front-end and back-end in web development lingo describe the code that is being run in a user client's computer versus the code that is being run over at the server respectively. In the illustration of how connecting to the database would work in Figure 21, front-end code would be loaded from the server to a web browser. The user could send requests from the browser to the server where back-end code would process the requests and run interactions between the application layer and the database.

After learning the basics of Angular and studying the server-client synchronization of MEAN the thesis author found out about an alternative solution called Meteor [40]. While MEAN uses individually developed frameworks together, Meteor differs in the

way that it has been conceived to be a singular software platform for data handling web applications. For example it handles much of the client-server interchange on its own and peculiarly incorporates the use of a MongoDB database natively. The opacity of the MEAN server-side functionality meant that Meteor's approachability seemed greatly preferable. Although a more recent technology than any of the parts in the MEAN-stack, Meteor's downsides, like possibly scarcier package libraries, were deemed insignificant enough and a switch into using Meteor instead of MEAN was made. Building a user interface in Meteor can be done in the aforementioned Angular, or by using alternative front-end toolsets React or Blaze [41]. While there was some experience on Angular already a change to Blaze was also made since it has been specifically developed as a "built-in reactive rendering library" for Meteor (as described by its own webpage) [41].

As such, the application would be written in the Meteor framework with a MongoDB database core and with Blaze as its front-end toolset. After studying the basics of Meteor the work on a proof-of-concept of the database system was started. The author also studied best practices on how to utilize automated testing during the development, but in the end it was decided that a basic development process should be properly learned before any useful automated tests could be deployed. Thus traditional manual debugging was to be used at least until it would get too unwieldy, even though automated testing is a core procedure in today's professional software development according to O'Regan [42].

5 Software Development

This chapter outlines the development process of the database software and details encountered problems, found solutions and decisions made during the actual programming work.

5.1 Setting Up The Application

When doing groundwork for the app a sensible file structure seemed important since the author was aware that poorly structured programs can easily become cumbersome when they grow larger. Fortunately for the author Meteor's guide page both explained Meteor's initial directories and had an example design pattern for the rest of the file tree [43]. The basic top level directories that were used were:

- **sipdata** - The root directory of the project. The abbreviation "sipdata" is from the words "silicon photonics database".
  - **.meteor** - Meteor creates this directory automatically. It stores Meteor's built-in logic and packages.
  - **client** - Code in this directory is not loaded on the server. In practice this folder contains imports (see below) from elsewhere.
  - **imports** - Meteor loads code from this directory only if it is called with a specific import-command. The Meteor guide recommends including all logic functionality the programmer writes in this directory, so that it can be loaded only when necessary - Meteor loads most code elsewhere no matter the situation.
  - **node_modules** - An automatically created directory for Node-modules. Meteor is built on the NodeJS technology, and any imported Node modules are stored in this directory. The modules are loaded when they are called with an import-command.
  - **server** - Code in this directory is not loaded client-side. Especially sensitive information like user account logic should be kept in a server-folder. In practice it contains imports like its client-directory counter-part.

The author's design (as influenced by the guide) was to have only single files inside the top-level client and server folders. Both contained only import commands to specify what code files were to be run in either front-end or back-end. Everything else that would be written by the author would go under the imports directory. The imports directory was divided into sub-folders as recommended by the Meteor guide:

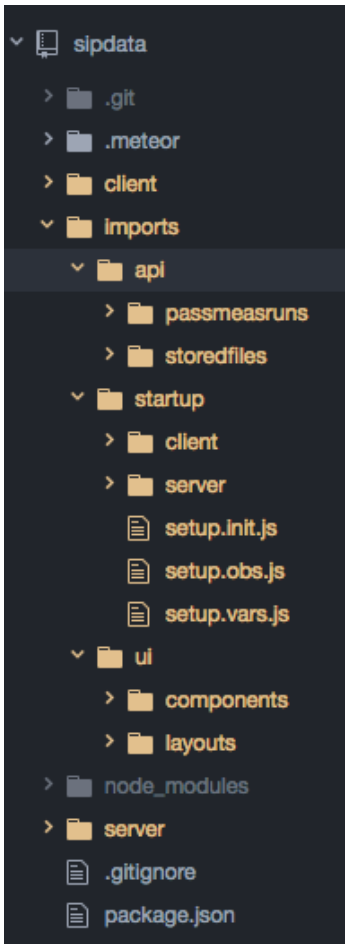


Figure 22: A screen capture of the application's final directory tree as shown in the editing program Atom. The "imports" folder is shown up until second subfolders as it is the most relevant directory. The "api" subfolder's name comes from "application programming interface" (API) which commonly describes functionality interfacing raw data with external programs. The root-level 'package.json' configuration file is created automatically by Meteor.

- **imports**
  - **api** - Contains code for handling data and defining the database structure, i.e. it contains the "soft" schema of the database.
  - **startup** - Contains initialization code that runs when the app is started.
  - **ui** - Contains code for how the user interface looks and functions.

This subfolder structure is not strictly mandatory since Meteor doesn't have internal requirements for these directories (unlike the top-level directories listed earlier). However as noted, this structure was recommended and the author agreed with the notion that dividing distinct parts of the program would be practical. In regards to the underlying MongoDB instance, the api-folder was very important. It contained all

collection definitions and the required fields in and relations between the collections. Also any user interaction (which were defined in the ui-folder) with the database would have to be validated by checks residing under the api-folder.

The final version of the project app had multiple further subfolders inside the three imports subfolders and the top-level client-directory ended up having a bit more than a single file in it, but the overall folder hierarchy described above was preserved throughout the work. A portion of the final directory tree is shown in Figure 22. The folder ".git" along with the file ".gitignore" seen in the figure relate to version handling and are not loaded with the application itself.

After creating an empty directory tree in the design outlined above, the first functionality that was worked on was a user account system. With the aid of a guide on the subject a basic account system was relatively easy to implement using two premade packages [44]. It was deemed important to have rudimentary user accounts in place from the beginning to avoid any security implications a later implementation could cause.

5.2 File Transfer Between Client and Database

With the app consisting of barely nothing more than the basic user account system, a larger effort was directed to develop functionality for file transfer between a client and the server. Since file transfer seemed like such an integral part of the project motivation it made sense to prioritize it early. It was thought that to achieve a good user experience uploading files should be as effortless as possible. A design solution included an upload area on an upload page which a client user could use to insert multiple files with either "drag-and-drop" or "click-and-select" methods.

A few Meteor packages for file transfer were considered for use. The package "files" by "ostrio" seemed like a good choice since it marketed itself as being a simple clutter-free file handling solution. However it lacked native support for MongoDB's GridFS technology, which allows files of any size to be stored. Although it's possible to manually implement GridFS in "files" according to the package's own guide [45], another package called "file-collection" by "vsivsi" turned out to be more readily suitable. Its page describes it as being designed for HTTP file transfer and for "efficient handling of millions of small files as well as huge files 10GB and above" [46]. According to the page its features are achieved because it integrates the GridFS technology with Meteor's native "Collection" data handling [46].

Implementing file uploading using the external "file-collection" package could be described as a small leap of faith. With little in-depth knowledge on how transferring files over a network worked an example code snippet written in another language, CoffeeScript, was transcribed to JavaScript. With no full understanding of the functionality of the code, this method required some time until a working file upload feature was in place. Even then some of the intricacies of the code used had to be studied much later. File download wasn't implemented at this stage, but "file-collection" had native functionality that made it a relatively easy addition later on.

To test the basic functionality created so far a test web page was also built. When

uploads were implemented the web user interface contained an upload area, or zone, and a list above it where all added files appeared. As is common with present day web apps, Meteor can populate a list of data items dynamically without the client having to refresh their web page. Utilizing user accounts, only logged in user clients could see any files uploaded. Creating the UI benefitted as much from external packages as the backend development, and "blaze-layout" and "flow-router" by "kadira" are just few examples of those that were used.

5.3 File Handling and Database Structure

After it was possible to upload files through a web browser, some work on how the data was handled was done. While MongoDB doesn't enforce using schemas like an RDBMS, "soft" schemas are still useful for creating links between data entities in the database. The collection schema that made most sense to prototype first was for storing results from optical measurements on silicon photonic components, since it was the research activity the thesis author had the most personal experience on.

A way to define schemas in Meteor could be done by using a package "simple-schema" by "aldeed". The package lets the developer to decide what fields are generated for all new documents in a collection, and which fields have to include initial values or are automatically filled (e.g. automatic timestamps). For the optical measurement collection, which was further specified to be for passive (non-electric) components only, fields initially defined included:

- insertion timestamp, auto-filled
- ID of the operator
- associated project
- project collaborator or customer
- component type
- chip ID

In addition to the above, fields to store all lab equipment measurement specifications and many IDs of related files such as measurement plans and lithography masks were defined or planned.

Since the optical measurements are often done in batches where e.g. dozens of similar components from a single chip are measured in the same fashion, it was clear that plenty of the measurement documents would have many exact same field values. This would cause unnecessary clutter, so the initial collection was split into two. (Passive) "Measurement files" stored uploaded result files and "measurement runs" collected all fields shared by the single measurements. A snippet of the code used to define the "run" schema in the api-folder is shown in Figure 23. Both collections also stored ID references to each other, so that a run document had an ID list of related result files and the file documents stored the ID of the run. Creating new documents

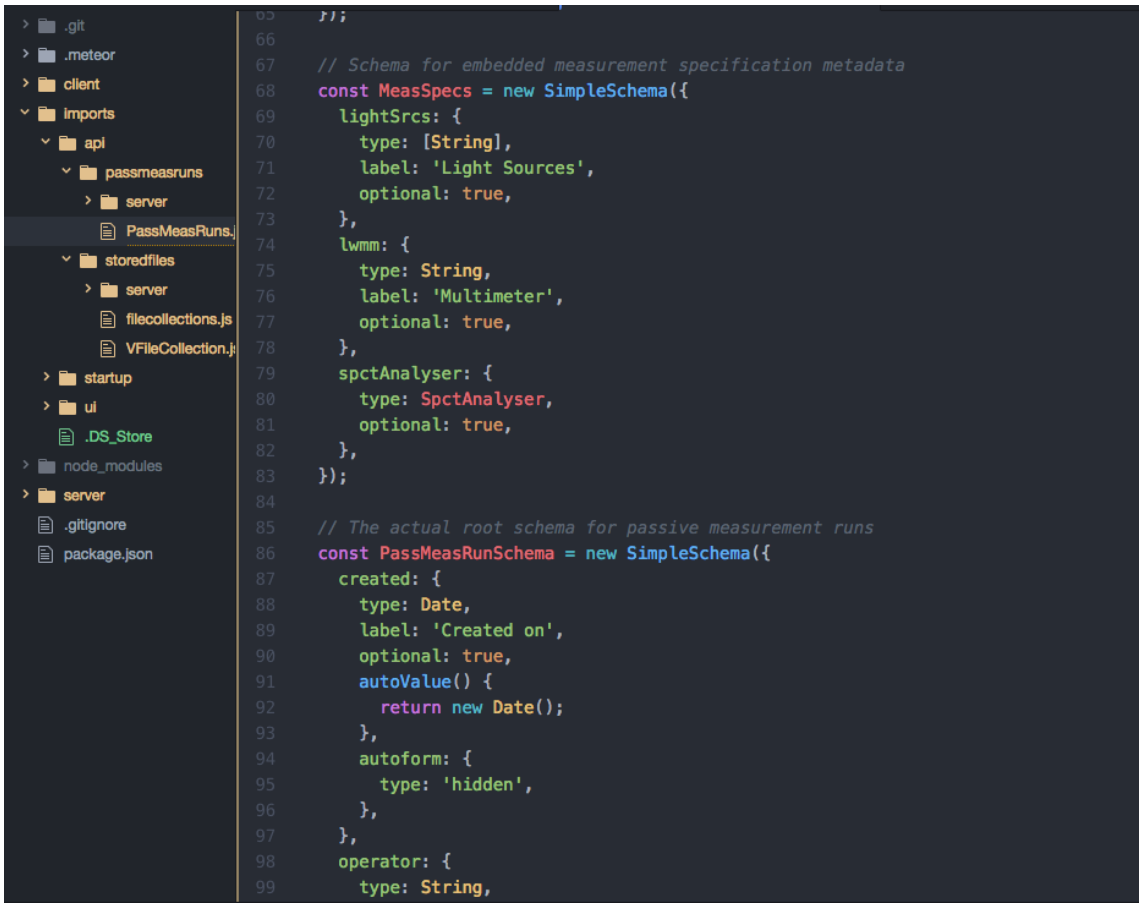


Figure 23: A snippet of the file "PassMeasFiles.js" opened in Atom. The file defines the schema for the metadata "run" documents, which link to the uploaded result files. The collection for results is created in "filecollections.js" seen in the directory tree on the left. The "MeasSpecs" schema definition is used as a "sub-schema" to embed grouped data units within the top-level schema "PassMeasRunSchema".

in the system required filling the shared fields and uploading results, after which the client user would press a submit button to confirm the creation of a new run document referencing all the related files.

In the test interface, adding new run documents was implemented by expanding the earlier file upload page. Using another package, "autoform", by "aldeed" a basic input form could be generated directly from the schema defined earlier. Also at this point the code base had been refactored to support multiple GridFS collections for uploaded files as well as moving initialization parts of the upload code into a new dedicated startup file. A download button was added which at first was hard coded for the single existing GridFS collection, but which was not much later generalized to work with new file collections as well. Pressing the button would automatically look up the desired file's ID and create a specific URL download link which would then be used to initiate the download.

After creating the prototype schemas a rudimentary search function was imple-

mented. The ability to search and sort documents was afterall one of the major motivations to develop the system. The search function used a simple text input which the server would read and use to find and retrieve only relevant documents to the user. In the initial state, the search function only queried associated project names, customer or partnership and insert timestamps. This text query feature was mostly done to test and learn basic search functionality. In a more complete version a faceted search function using both text input and category tags was planned.

Client-side, it was intended that no users should be able to see documents they hadn't got the permission to view. Initially for VTT internal use this would only mean files that were uploaded by another user client but not yet confirmed. Meteor uses an innate technology called publishing which makes user permissions relatively easy to implement, and which was actually the central technology utilized in the search function already in place. The developer can specify how a collection publishes itself, and when a user client tries to subscribe to a collection, its publish rules are automatically checked. According to the extent of the specifications fulfilled, a full or limited set of privileges can be granted. For example an internal user would possibly be able to view all data fields but be unable to modify any unless they were the document author themselves, and an external customer client would perhaps see only limited data. No complex flagging like this was deemed important to be implemented at this point though.

An issue with the new search functionality was that if a user had uploaded files but hadn't yet created a related run document for them (and thus confirmed the whole transaction) another user could have already queried the files in the search interface. To avoid inconsistencies this could cause "publish-composite" package by "reywood" which allows for "chaining" of publications was used. It enabled a functionality where user clients subscribed first to a measurement run document after which they automatically subscribed to any relevant results referenced by the run. This way user clients couldn't see result files which another user hadn't committed since the files didn't have an associated run document yet. The exception was that if the user client was themselves authoring a new run document, they would directly subscribe to the files they had uploaded (so they could review what they were about to submit).

File removal was also added during the same time period as the search function. The user client had to check a marker box on each of the files they wanted to remove and then confirm the action by clicking a delete button. The function would look up the IDs of the selected files and use them to find and remove the correct files from the database. Only deletion of files waiting for commit by their author was made possible. This was because it was not decided if and to what extent administrational approval was to be required to delete files after the files would be committed to the system.

The two-step file storing (upload and confirmation) meant that in the case a user client would log out in between the steps for any reason, some hidden files would be left behind and clog up the database. To solve the issue, a pruning check on user logout was implemented to look up if any files authored by that user lacked a required reference value. For example in the case of the optical measurements

collection, result files with no value for a related run document would be found and deleted. The same check also pruned all data chunks left by an unfinished upload process if a user client logged out before a file was fully transferred. In longer use this kind of pruning feature would perhaps require some fine tuning so it didn't search through all uploaded data on every user logout.

6 Results

This chapter outlines the features present in the database system at the time the system development project ended. Plans for future development are also described here. Project conclusions are presented in Chapter 6.

6.1 Features

The most properly developed functionality in the application layer is file transfer via HTTP uploads and downloads. Batches of files can be uploaded simultaneously either by dragging and dropping a selection into a specified upload space, or clicking the space and choosing files manually. Downloading is presently possible one file at a time. Uploading has basic failsafes regarding possible junk data. If a user client logs out for any reason before an upload is complete or committed, the system will automatically detect and delete the leftover data chunks and files. A known uploading issue is that if the browser window is refreshed after a file upload but before an associated run document is created, the run can't be committed and the user has to start over and reupload the files.

A prototype of a scheme to store passive measurement batches ("runs") is currently in place. The scheme uses two collections where one stores metadata related to the whole batch (equipment specifications etc.) and the other stores uploaded result files. The collections are linked by the run document storing all related result IDs and the result documents storing the related run ID. The system has underlying code built for handling many GridFS collections for file storage. In practice the system was tested at most with two GridFS collections, with a test-purpose dummy collection set up besides the measurement uploads collection. A search function uses a freeform text input and uses it to filter files from the database. The query is presently applied to fields containing associated project, timestamp and project customer or partner.

A user account system exists which allows only logged in users view anything else than a front page. A client that is not logged in trying to access other pages is automatically redirected to the front page. The redirect fires also when a user logs out while viewing a content page. The account system is in other ways very rudimentary, e.g. it doesn't yet require any authentication on new user accounts.

The user interface is largely a test piece. A front page has a button for login or creating an account. There are three content pages that are accessed via a navigation bar. These pages are shown in Figures A1-A4 in Appendix A. The first page (Figure A1) is for passive measurement run uploads. The page contains a form where a user fills in fields containing e.g. the project name associated with the measurements and measurement specifications. Below the form is an upload area. Added results appear dynamically in a list above the area, with a live percentage number on each entry indicating the progress of its upload. Once all necessary information is in place the user presses a submit button that confirms the transaction and the user is redirected to a search page. In the uploads list there are buttons which are used for downloading and removing files. A placeholder button for flagging files as reference results is also present, but its functionality isn't implemented.

The second content page (Figures A2 and A3) is for uploading files into a test collection. This was created and used to test the system with more than one GridFS collection. The third page (Figure A4) is used for searching files. A text input populates a found files list automatically according to the search query, as shown in Figure A5. The input text is matched against the timestamp, project and customer or partnership fields of the files in the database. The list looks identical to those generated in upload sections, but the file deletion button intentionally does not work here.

While the development of the database system was done on the author's work station, in real use the system would be deployed on a server. There were talks about either acquiring a physical server machine for the research group or renting a virtual cloud server. In the end a VTT-contracted virtual server was available with no additional cost and as such it was used to host the system. During development the system had used an internal Meteor test database. However, when deployed onto a server into actual use Meteor requires a connection to a stand-alone MongoDB instance. The linking itself is relatively simple, Metor only needs to be set up with a correct "MONGO_URL" environmental variable and it automatically connects to the database [47]. Setting up the server to have HTTP connections from elsewhere in the network reach the system required some time but a working state was reached. The system was proven to work with multiple concurrent user clients connecting via VTT's intranet. A researcher could also access the underlying database using his own Python-language code and as such bypass the existing application layer. This is considered positive since it allows more flexibility in how the system could be used.

The development of the system would not have been possible without pre-made modules and packages created by the Meteor development team and community. The packages enabled e.g. fast implementation of user accounts with password encryption, easy functions to control HTTP data transfer, the creation of schemas and much more. The list of added packages is shown in Figure 24.

6.2 Development Plans

The database system in its present state is mostly a proof-of-concept version containing basic functionalities. If it was to be developed further the next steps would be implementing proper data structure. While it currently contains only a collection for passive optical measurement sets and another collection for related results, the design vision was for it to store numerous interrelated collections. Possible collections could include some listed below:

- **collaborators** - Relevant information of customers and partnerships
- **projects** - Relevant details of specific projects
- **masks** - Mask files used in lithography during chip manufacturing
- **design documents** - Component design information and description
- **process description** - A short overview for the manufacturing process

- **instructions** - Instruction files for manufacturing processes and measurement
- **wafers & reticles & chips** - One or multiple collections for storing information of the physical chips and their associated reticles and wafers
- **cleanroom measurements** - One or multiple collections for measurements done in clean room environment
- **active component sets & results** - Electro-optic measurement results

The collections would have relations (implemented by storing "related ID" lists) between them, which would serve a properly fleshed out search feature.

The present search functionality uses only a text query which is matched against a few fields across the collections. It would have to be expanded to achieve a proper information retrieval feature. The planned search feature would probably use principles of faceted search, which are outlined by Tunkelang [48]. The user would be able to still use a text query, but also filter the results according to plenty of relevant categories including specific collections, date, project, customer, component type etc. The interface could be similar to search pages found in many online stores.

```
# Added packages:

meteortoys:allthings          # Mongol and JetSetter developer tools, a debugOnly package

kadira:flow-router            # Client-side router
arillo:flow-router-helpers    # Routing helpers
kadira:blaze-layout           # Layout manager
raix:handlebar-helpers        # Helpers for sessions and collection in template environment
zimme:active-route            # Helpers to determine if a route/path is or isn't currently active

aldeed:simple-schema          # Create matching schemas for collections
aldeed:collection2            # Attach schemas on collections
aldeed:autoform               # Create forms automatically for defined schemas

check@1.2.3                   # Functions for checking the types and structures of variables
matb33:collection-hooks       # Many server side before and after hooks
reywood:publish-composite     # Publish documents jointly from multiple collections

vsivsi:file-collection        # Uploading and Mongo GridFS support
benjaminrh:jquery-cookie      # Needed for HTTP file uploads

accounts-ui@1.1.9             # UI elements for user account stuff
accounts-password             # User accounts with passwords
differential:event-hooks      # Adds various hooks for user-triggered events

stylus@2.513.5                # CSS pre-processor
semantic:ui                   # UI element library
less@2.7.5                    # Semantic uses Less and -
juliancwirko:postcss          #   - PostCSS
fabienb4:autoform-semantic-ui # Semantic UI styling for autoform
```

Figure 24: A list of external packages used in the project, with comments clarifying each of their purpose. The image does not show standard Meteor packages. In general the naming convention for the packages is "developer:package".

Before the system could be used its user accounts should be more robust. At the very least new accounts should require an administrational approval. Later on different user permissions could be categorised, e.g. limited administrational privileges related to data in certain research area etc. The user interface would need an overhaul and lots of new pages to handle adding different documents, as well as the faceted information retrieval discussed earlier.

In further off plans, there were discussions of the system being able to do rudimentary analysis and graphical presentation, as well as permitting limited external access. An example of the former would be calculating average transmission power from optical measurements related to a reference result. The computed average could then be plotted against wavelength in an image describing transmission loss. External access into the database would let VTT collaborators like customers or university partners etc. find and use some of the collected data. However, access to the system through Internet would need a lot more safety development and testing as well as probably an approval and moderation from the company's responsible IT personnel.

7 Conclusions and Future Work

The database system at this point is a working proof-of-concept of a basic web data storage application. Relatively early during the development process it started to seem doubtful that a fully functional database management system could be conceived in the span of this thesis. However, the application that was realized is attempted to be built to be as robust and easily extendable as possible, so if future development came to be it should be able to work as groundwork for a proper software. Data transfer is the most complete full feature, and since any kinds of files can be transferred its code should be reusable in many cases.

The data structure was developed only into a fairly rudimentary state, but implementing new data and GridFS collections should be easy enough reusing the created prototypes. The faceted search functionality is probably the largest feature that was long envisioned as reaching a more fleshed out state by the project end but which was left lackluster. Although, some functionality of how to filter data from the collections were written when the test search was created which should be reusable in a more robust feature.

Developing the user interface often seemed like a bit of a distraction from the server code, but on the other hand developing the back-end often necessiated more features in the UI. This is one reason why the development of a faceted search and a larger collection library didn't fit into the project timespan. Also in retrospect, some unimportant features in the test UI took probably more development time than they should have, such as percentage indicators while uploading files.

Another retrospection related to the UI concerns the front-end framework that was used. The framework Blaze was a practical choice in that it is developed for use with Meteor. However, if another developer continued the project later on, the more popular toolsets React or Angular could make the possible transition easier. On the other hand the present user interface was created mostly to serve testing purposes as was discussed earlier. As such a new developer responsible for a follow-up project would need to overhaul the UI in any case, and could rewrite it wholly with their chosen toolset without too much lost effort.

A lot of project time was initially used to learn basics of web development and finding suitable tools and it is admitted that the project's early phase somewhat lacked direction. There was much early focus on theory of databases and web applications while afterwards a more practical approach revealed a lot of the early study went outside the scope of the project. Meteor was found to be a suitable framework and it seems sensible that it was used instead of the more complex MEAN-stack even if the switch between the two was made relatively late. Given the broadness of the project, developing the UI could perhaps have been its own project or made by another person altogether. In retrospect it seems unfortunate that there were no readily available senior developers for guidance since many issues that took hours and maybe days of development time were relatively simple to solve after the necessary knowledge was found.

On the whole, a more modular approach could have been smarter. The application was being designed to handle as many use cases as possible, but if the project was

started over the author would probably develop a web portal solely for searching information and downloading files. All uploads and writes could be then handled case by case. Already during the project timespan there was talk about researchers using their own scripts to interact with the database and also about a software block made with LabView to upload measurement files. A potential downside with this approach would be that the system could turn out complex and cumbersome in the long run, but it would still probably be a more realistic way to get a self-made DBMS to work in a research environment.

The thesis project's initial goals demanded perhaps too much from a student with no background in databases or web development. On the other hand, after a reassessment of project aims success was found with the realization of a working proof-of-concept database system which can pave the way of a later more robust implementation. The thesis author also certainly learned much about the subject fields.

After this thesis project the database system development was paused. To the author's understanding this was in part due to talks of a larger scale process handling or database software plans, with larger scope than the silicon photonics research area. If development of this thesis project is continued, the author's opinion is that the most robust feature in place is the MongoDB database server. The Meteor application is not necessary for the database's basic functionality or network connectedness. The application layer can be re-developed or continued from the author's Meteor code at new developers' own discretion. If control over data structure is deemed important, a new MongoDB version includes a "JSON schema" feature [49]. Using the JSON schema would allow leaving out data structure definitions in the application layer, simplifying it and likely causing less conflicts if more modularity was adopted.

References

[1] Vivien, L. and Pavesi, L. *Handbook of Silicon Photonics,* Boca Raton: Taylor & Francis Group LLC, 2013, p. Preface..

[2] Bergman, K. et al. *Photonic Network-on-Chip Design,* New York: Springer Science+Business Media, 2014.

[3] Inniss, D. and Rubenstein, R. *Silicon Photonics: Fueling the Next Information Revolution,* Cambridge, United Kingdom: Elsevier Science, 2016.

[4] Aalto, T. Comments from the thesis supervisor, 8th July 2017.

[5] Homepage of the project MIREGAS. Referenced on 27th July 2017. Available at http://www.h2020-miregas.eu/H2020-MIREGAS/Home.html.

[6] Homepage of the project BiopsyPen. Referenced on 27th July 2017. Available at http://www.biopsypen.eu/.

[7] "VTT Silicon Photonics Slide Presentation (Santa Clara 2016)" slides. Referenced on 12th June 2017. Available at http://www.vtt.fi/sites/siliconphotonics/download.

[8] Bogaerts, W. and Selvaraja, S. K. Chapter 13 in *Silicon-on-Insulator (SOI) Technology - Manufacture and Applications,* Cambridge, United Kingdom: Elsevier, 2014, p. Chpt 13.

[9] Aalto, T. et al. *Total internal reflection mirrors with ultra-low losses in 3 μm thick SOI waveguides,* Proceedings of SPIE Vol. 9367, 2015, p. 93670B.

[10] Subbaraman, H. et al. *Recent advances in silicon-based passive and active optical interconnects,* Opt. Express, vol. 23, no. 3, p. 2487, Feb. 2015.

[11] Aalto, T. and Harjanne, M. and Cherchi, M. *VTT's micron-scale silicon rib+strip waveguide platform,* Proceedings of SPIE Vol. 9891, 2016, p. 98911G.

[12] Roelkens, G. and Van Thourhout, D. Chapter 3 in *Silicon Photonics II: Components and Integration,* Berlin, Germany: Springer, 2011, p. Chpt 3..

[13] Zhu, L. and Yang, W. and Chang-Hasnain, C. *Very high efficiency optical coupler for silicon nanophotonic waveguide and single mode optical fiber,* Opt. Express, vol. 25, no. 15, p. 18462, Jul. 2017.

[14] Cherchi, M. et al. *Dramatic size reduction of waveguide bends on a micron-scale silicon photonic platform,* Opt. Express, vol. 21, no. 15, p. 17814, Jul. 2013.

[15] Bogaerts, W. et al. *Silicon-on-Insulator Spectral Filters Fabricated With CMOS Technology,* IEEE J. Sel. Top. Quantum Electron., vol. 16, no. 1, pp. 33-44, 2010.

[16] Wikimedia, a ring resonator coupling coefficients image. Referenced on 12th June 2017. Available at https://commons.wikimedia.org/wiki/File:CouplingCoefficients.png.

[17] Wikimedia, a ring resonator coupling image. Referenced on 12th June 2017. Available at https://commons.wikimedia.org/wiki/File:RingResonatorCW.png.

[18] Wikimedia, a resonant wavelengths image. Referenced on 12th June 2017. Available at https://commons.wikimedia.org/wiki/File:MultipleResonances.png.

[19] Deen, M. J. and Basu, P. K. *Silicon Photonics: Fundamentals and Devices,* Chichester, United Kingdom: Wiley, 2012.

[20] Wikimedia, an illustration of an AWG. Referenced on 12th June 2017. Available at https://commons.wikimedia.org/wiki/File:Arrayed-Waveguide-Grating.svg.

[21] Harrison, G. R. *The Production of Diffraction Gratings: II. The Design of Echelle Gratings and Spectrographs,* Journal of the Optical Society of America, vol. 39, no. 6, p. 413, 1949.

[22] Dabos, G. et al. *Thick-SOI Echelle grating for any-to-any wavelength routing interconnection in multi-socket computing environments,* Proceedings of SPIE Vol 10109, 2017, p. 1010914.

[23] Ryckeboer, E. et al. *CMOS-compatible silicon nitride spectrometers for lab-on-a-chip spectral sensing,* Proceedings Volume 9891, Silicon Photonics and Photonic Integrated Circuits V, SPIE Photonics Europe, 2016, p. 98911K.

[24] Homepage of the RAPIDO project. Referenced on 28th of July 2017. Available at http://www.rapido-project.eu/index.html.

[25] Ullman, J. D. and Widom, J. *A First Course in Database Development,* Harlow, England: Pearson Education Limited, 2014.

[26] Codd, E. F. *A Relational Model of Data for Large Shared Data Banks,* IBM Research Laboratory, 1970.

[27] Dietrich, S. W. and Urban, S. D. *Fundamentals of Object Databases: Object-Oriented and Object-Relational Design,* Morgan & Claypool, 2011.

[28] Vaish, G. *Getting Started with NoSQL,* Birmingham, England: Packt Publishing, 2013.

[29] Foster, E. C. and Godbole, S. V. *Database Systems: A Pragmatic Approach,* New York: Springer Science+Business Media, 2014.

[30] Tiwari, S. *Professional NoSQL,* John Wiley & Sons, Incorporated, 2011.

[31] Elmasri, R. and Navathe, S. *Fundamentals of Database Systems,* Harlow: Pearson Education Limited, 2014.

[32] Darmawikarta, D. *Oracle SQL: A Beginner's Tutorial,* Brainy Software Inc., 2014.

[33] Churcher, C. *Beginning Database Design: From Novice to Professional,* New York: Springer Science+Business Media, 2012.

[34] Mason, R. T. *NoSQL Databases and Data Modeling Techniques for a Document-oriented NoSQL Database,* Proceedings of Informing Science & IT Education Conference (InSITE), vol. 3, issue 4, 2015, pp. 259-268. Retrieved on 21st Oct 2017 from http://Proceedings.InformingScience.org/InSITE2015/InSITE15p259-268Mason1569.pdf.

[35] Changlin, H. *Survey on NoSQL Database Technology,* Journal of Applied Science and Engineering Innovation, vol. 2, no. 2, pp. 50-54, 2015.

[36] Screenshot of an example JSON on Wikipedia. Referenced on 21st Oct 2017. Available at https://en.wikipedia.org/wiki/JSON.

[37] Hows, D. and Membrey, P. and Plugge, E. *MongoDB Basics,* New York: Springer Science+Business Media, 2014.

[38] Grant, A. *Beginning AngularJS,* New York: Springer Science+Business Media, 2014.

[39] Mean.io homepage. Referenced on 14th Feb 2017. Available at http://mean.io.

[40] Meteor homepage. Referenced on 14th Feb 2017. Available at http://meteor.com.

[41] Blaze introduction, a web page. Referenced on 16th Feb 2017. Available at http://http://blazejs.org/guide/introduction.html.

[42] O'Regan, G. *Introduction to Software Quality,* Springer International Publishing Switzerland, 2014.

[43] Meteor Guide: Application Structure, a web page. Referenced on 23rd Mar 2017. Available at https://guide.meteor.com/structure.html.

[44] Meteor Blaze tutorial, a web page. Referenced on 24th Mar 2017. Available at https://www.meteor.com/tutorials/blaze/creating-an-app.

[45] Meteor 'files' package documentation, a web page. Referenced on 27th Mar 2017. Available at https://atmospherejs.com/ostrio/files.

[46] Meteor 'file-collection' package documentation, a web page. Referenced on 27th Mar 2017. Available at https://atmospherejs.com/vsivsi/file-collection.

[47] Meteor Guide: Deployment and Monitoring, a web page. Referenced on 27th Apr 2017. Available at https://guide.meteor.com/deployment.html.

[48] Tunkelang, D. *Faceted Search; Synthesis Lectures on Information Concepts, Retrieval, and Services*, Chigago: Morgan & Claypool Publishers, 2009.

[49] MongoDB 3.6 Empowers Enterprises and Developers to Move at the Speed of Data, a web page. Referenced on 15th Nov 2017. Available at https://www.mongodb.com/press/mongodb-36-empowers-enterprises-and-developers-to-move-at-the-speed-of-data.

A User Interface

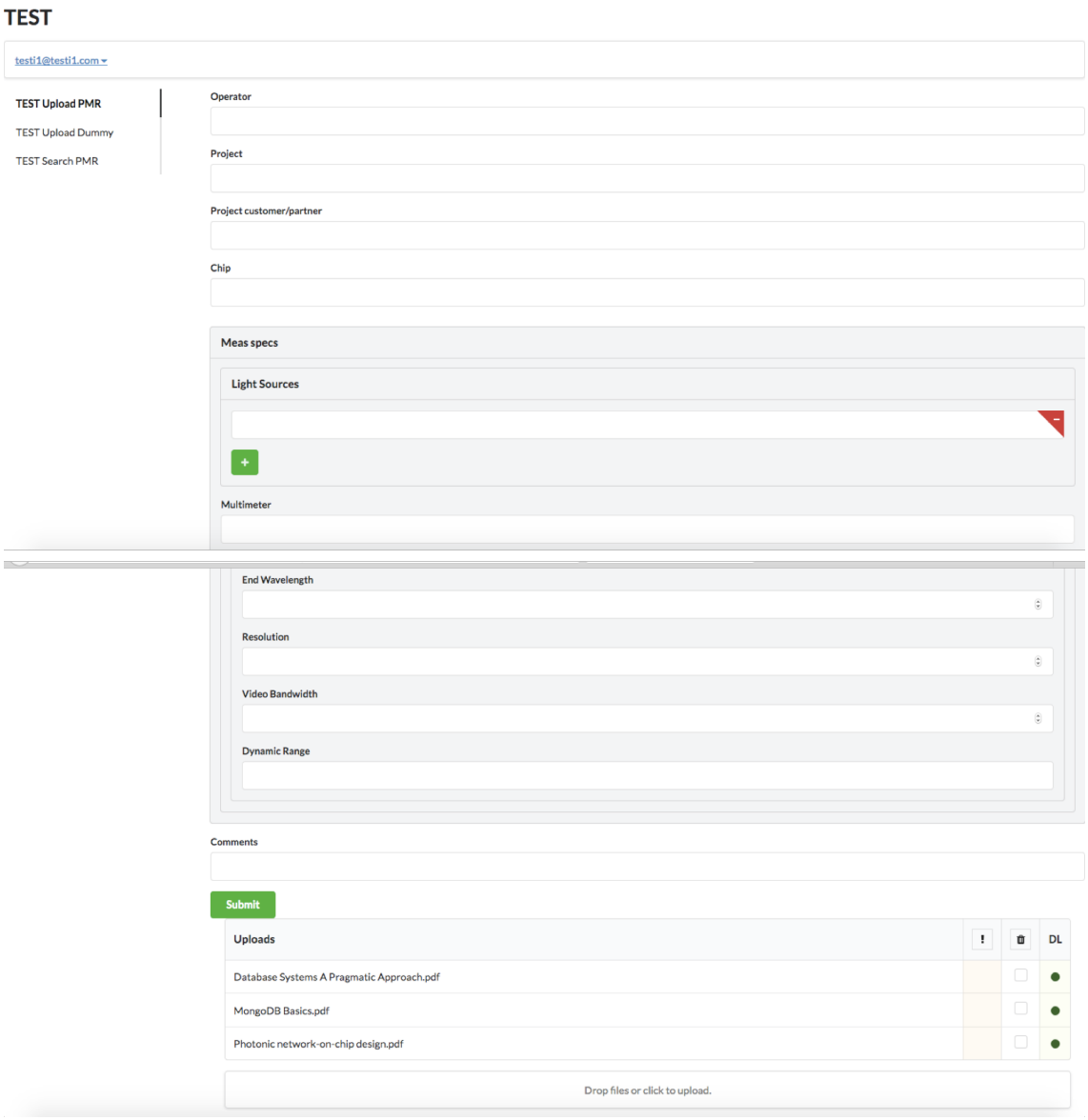


Figure A1: Two screen captures from an upload page for measurement result files. Some test files have been uploaded to show how a list of uploaded files is shown.

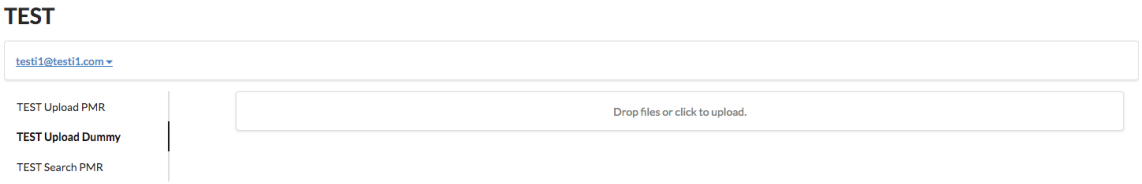


Figure A2: A simple file upload page for testing the use of more than one GridFS collection concurrently.

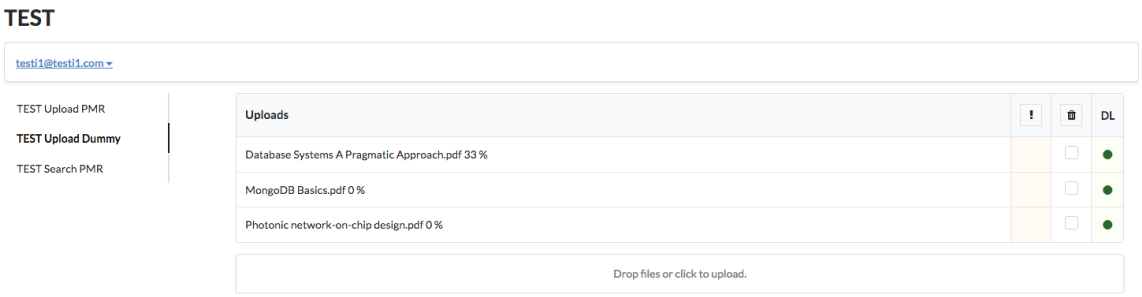


Figure A3: The same upload page as in Figure A2, but with an upload of three files in progress.

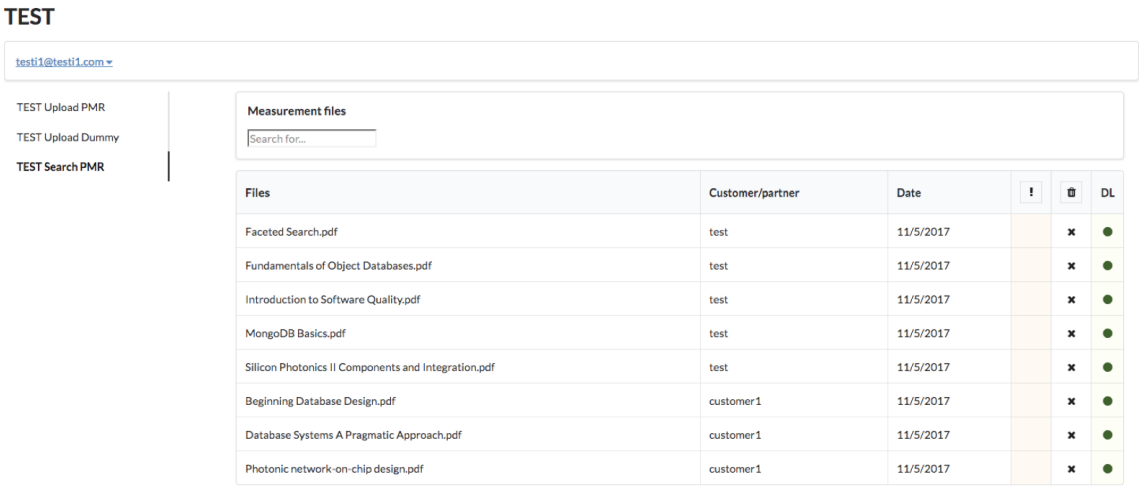


Figure A4: A search page.

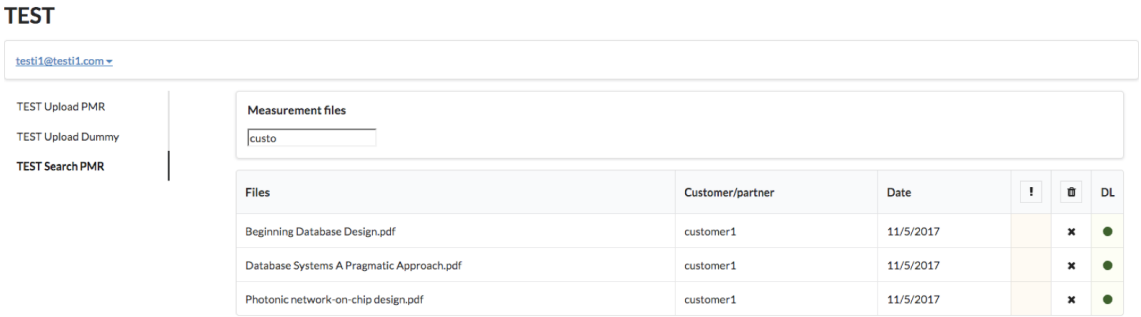


Figure A5: The search page showing only results relevant to the written search text.